goTenna SDK Documentation

Release 0.12.5

goTenna Inc.

Dec 17, 2018

CONTENTS

CHAPTER

ONE

INSTALLING

The goTenna SDK is a python setuptools project. It is distributed as a Python wheel which can be imported with `pip`: `pip install /path/to/goTenna/gotenna-x.y.z-py2.py3-none-any.whl`.

1.1 From An SDK Distribution

If you are installing from a zipped SDK distribution provided by goTenna, follow the instructions to install Python and required packages. When you run `pip`, the egg file will be in the same directory as this document. You can run `pip` like this: `pip install gotenna-x.y.z-py2.py3-none-any.whl`. If `pip` is not installed on your system as an executable (for instance, on Windows) you can invoke it through Python: `python -m pip install gotenna-x.y.z-py2.py3-none-any.whl`

1.2 Python and Required Packages

The system on which you wish to use the goTenna SDK, either for development or production, must have Python 2.7, the `setuptools` module, the `pip` module, and the `wheel` module installed. For instructions on installing Python on your system, see https://www.python.org/ under Downloads. Python may also be available through your system package manager, if you are on Linux or OSX. For instructions on installing `setuptools`, see the setuptools documentation. While any Python installation after 2.7 should come with `pip` and `wheel`, if they are not installed see the pip documentation to install `pip` and then type `pip install wheel` or `python -m pip install wheel` into a command prompt to install `wheel`.

1.2.1 Frequent Problems Installing Python

- On OSX, if the system installation of Python2.7 is used, some Python packages are installed in ways that `pip` cannot override. Installing packages like `setuptools` and `wheel` will produce errors about permissions even when `sudo` is used. To get around this, invoke `pip` with the `--ignore-installed` flag so the system packages in question will not be updated. For instance, to install `setuptools` and `wheel` on such a system, invoke the command `sudo -H pip install --ignore-installed setuptools wheel`
- On Windows, if Python is installed on a per-user basis, the Python interpreter `Python.exe` will not be in the system path, and running commands such as `python.exe -c "print('hello, world')"` will fail because the system cannot find the interpreter. To solve this, either add Python to the system path or invoke the interpreter with its full path. For instance, if Python is installed in `C:\Python27`, you can invoke the interpreter with the command `C:\Python27\python.exe -c "print 'hello, world!'"`

1.3 goTenna SDK Installation

Once your system has Python, `pip`, `setuptools`, and `wheel` locate the goTenna SDK wheel on your filesystem (it will be called something like `goTenna-0.1.0-py2.py3-none-any.whl`; the `0.1.0` part is a version and may change) and run `pip install` on it. For instance, if `pip` is in your path (which it should be on most installations) and you are in the directory containing the wheel, the command is `pip install goTenna-0.1.0-py2.py3-none-any.whl`.

At this point, the module will be copied to your Python's system path where it keeps all its other modules. If you already have a module with the same name and an older version, the new module will be installed over it.

In addition to copying the egg, `pip` will download and install the dependencies we require. Right now those dependencies are pyserial, at or above version 2.6; cryptography; and six If `pip` will not be allowed to access the Internet during installation, please install these modules manually before installing the goTenna SDK.

Once you have run `pip install`, the goTenna SDK is installed on your system. If you wish to test, run the command `python -c "import goTenna; print(goTenna.api_version)"`. This should print out the version of the SDK that you just installed.

note Depending on the way you installed Python, you may need administrator privileges or sudo access to install the goTenna SDK. If you do not have administrator privileges, see *Using virtualenv*.

1.4 Using virtualenv

virtualenv is a tool used to create isolated Python environments so different applications with different requirements can be run on the same system without conflicting. While goTenna does not require the use of `virtualenv`, it does support it. To install, simply create and activate the virtualenv and then run `pip install`, as detailed above, in the virtualenv.

1.5 Frequent Problems Installing the goTenna SDK

- On Linux, `openssl-devel` (or `libssh-dev` depending on distro) and `libffi-dev` are required to build the cryptography module we use.
- On Linux, on at least Debian-based distributions, installing `cryptography` and `cffi` through the system package manager can cause conflicts with the versions that `pip` attempts to install, which reproduce as `pip` segfaulting while installing our wheel. To workaround this, uninstall the system packages `python-cryptography` and `python-cffi-backend` (or `python3-cryptography` and `python3-cffi-backend` on a multi-python system if you want to use python3) before installing the wheel.
- On Linux, the modem manager deamon might try to capture the goTenna USB link making the SDK connection fail with a timeout error. In this case, copy the `77-gotenna.rules` in /etc/udev/rules.d for the modem manager to ignore the goTenna devices.

CHAPTER

TWO

GETTING STARTED

The goTenna SDK is designed to allow a developer to quickly create software that communicates with goTennas via USB.

The primary interface to the SDK is the *goTenna.driver* module. It contains the *goTenna.driver.Driver* class, which is a threaded, event-driven interface to the SDK.

2.1 SDK Tokens

You should have been provided with an SDK token along with this SDK. This token is a string that is given to *goTenna.driver.Driver* when it is instantiated, and ensures that the applications you make with this SDK cannot interfere with applications made by other people using the SDK. goTennas using different SDK tokens cannot communicate with each other.

2.2 GIDs

GIDs are identifiers for users of goTennas. They are associated with a person rather than a device. Each device stores the GID of its current user, and uses this GID to determine if a private message is meant for the current user. The GID is passed in to the SDK either when a *goTenna.driver.Driver* object is created or by the *goTenna.driver.Driver.set_gid()* method.

A convenience class *goTenna.settings.GID* is provided to associate a GID and name. Methods of the SDK interacting with GIDs typically take and return instances of this class.

2.3 Event, Method, and Acknowledge Callbacks

Because the SDK runs in its own thread, it communicates back to the main application with callbacks. There are three types of callbacks: the *event callback*, which is associated with the `Driver` instance; the *method callbacks*, which are associated with individual method calls; and the *acknowledge callbacks*, which are associated with individual messages sent via the goTenna network.

The event callback is passed as an argument to the *goTenna.driver.Driver* constructor. It has this form:

event_callback(*event*)

> **Parameters** **event** (`goTenna.driver.Event`) – The event that just happened. The `Event` class contains a field called *goTenna.driver.Event.event_type* which can be used to determine the type of event that occurred. Depending on which event occurred, other fields in the `Event` object will be present.

This callback is used to inform the application of the status of the SDK. It is called by the SDK when any significant event occurs: a device is connected or disconnected, a message arrives, or there is new status data from the device.

> **note** The event callback runs in the context of the SDK thread.

The method callback is passed as an argument to method invocations on `goTenna.driver.Driver`. They have this form:

`method_callback`(*correlation_id*, *success=None*, *results=None*, *error=None*, *details=None*)

Parameters

- **`correlation_id`** (*`uuid.UUID`*) – A UUID that can be used to associate callback invocations with the method that invoked them. This can be used to have a generic method callback know when a specific method invocation has completed. This UUID is the same one that is returned by the method invocation to which this callback was passed.
- **`success`** (*`bool`*) – `True` if the method call succeeded; `None` or `False` otherwise.
- **`results`** (*`object`*) – If the method call has results to return to the main application, they will be in this argument. The type and contents of the results depends on the method being invoked, and methods that return results specify what will go into the callback. If there are no results, or if the method call failed, this will be `None`.
- **`error`** (*`bool`*) – `True` if the method call failed; `None` or `False` otherwise.
- **`details`** (*`dict`*) – If the method call failed, this parameter will include a dictionary with details about the failure. It will have two keys: `code`, containing an instance of `goTenna.constants.ErrorCodes`, and `msg`, containing a human readable description of the error. If the method succeeded, this will be `None`.

Method callbacks are used to allow method calls such `goTenna.driver.Driver.send_broadcast()` to return immediately and not block execution of the main thread. Method callbacks are passed in to any method of the driver that requires communication with the remote device. The callback is called when the method is done, whether it succeeded or failed.

> **note** The method callback runs in the context of the SDK thread.

Acknowledge callbacks are passed as arguments to `goTenna.driver.Driver.send_private()`, `goTenna.driver.Driver.invite_to_group()`, and `goTenna.driver.Driver.add_group()`. They have this form:

`acknowledge_callback`(*correlation_id*, *success*)

Parameters

- **`correlation_id`** (*`uuid.UUID`*) – A UUID that can be used to associate callback invocations with the method that invoked them. This is the same value returned by the method invocation to which this callback was passed, and the same correlation ID passed to the method callback for that invocation.
- **`success`** (*`bool`*) – `True` if the message was acknowledged by the recipient, `False` if enough time has passed that it likely never arrived.

> **note** A positive acknowledge callback means that the destination goTenna device received the message. It does not mean that the message arrived at whatever phone or device is connected to the goTenna; if the goTenna is not connected to a device, the message will still be acknowledged

> **note** The acknowledge callback runs in the context of the SDK thread.

2.4 Connecting to a goTenna

A started instance of *goTenna.driver.Driver* is always either connected to a device, or searching for a device to connect to. We determine the presence of a device by scanning the USB devices that present themselves as serial consoles, and finding any with the VID `0x1fc9` and PID `0x8181` for PRO or PID `0x8182` for Mesh. If your goTenna is not connecting, you can diagnose OS-level issues by checking for the presence of a USB device with these IDs.

If such a device is found, the driver automatically indicates its presence to the application by invoking the event callback with event type *goTenna.driver.Event.DEVICE_PRESENT*. If the driver is configured, it will then connect to the device, and invoke the event callback with event type *goTenna.driver.Event.CONNECT*.

To easy situations where multiple driver instances are running at the same time, the driver can be configured to connect to only devices of a certain type (for instance, goTenna Pro vs goTenna Mesh) or of certain serials. These limitations are specified either when the driver object is initialized, using the `device_blacklist`, `device_whitelist`, and `device_types` arguments to *goTenna.driver.Driver.__init__()* or by modifying *goTenna.driver.Driver.whitelisted_devices*, *goTenna.driver.Driver.blacklisted_devices*, or *goTenna.driver.Driver.accepted_device_types*. Connected devices that do not match the restrictions in those three attributes will not appear in device present events, and the driver will not connect to them manually.

To request the driver connect to a specific device (in terms of the filesystem path, e.g. `/dev/ttyACM0` or `COM15` at which it appears), use *goTenna.driver.Driver.connect()*. To prevent the driver from connecting in any other way, the parameter `do_automatic_connect` in *goTenna.driver.Driver.__init__()* or the method *goTenna.driver.Driver.allow_automatic_connection()* can be used.

To manually request a disconnect, the method *goTenna.driver.Driver.disconnect()* can be used.

2.5 Configuring a goTenna

The *goTenna.driver.Driver* must be configured before it will connect to a goTenna device. It requires the application to specify a private GID and (for a goTenna Pro) RF settings..

Both of these may be either configured while instantiating an instance of the driver through *goTenna.driver.Driver.__init__()* or by calling the setter methods *goTenna.driver.Driver.set_gid()* and *goTenna.driver.Driver.set_rf_settings()* respectively. Invoking any of these methods when the driver is connected will cause the driver to disconnect and reconnect. Invoking any of these methods with `None` as the argument will clear the configuration, and the driver will not connect to a goTenna until the configuration is set.

To check whether a *goTenna.driver.Driver* instance is fully configured and ready to connect, use the property *goTenna.driver.Driver.can_connect*.

2.6 Encryption and Storage

The goTenna SDK offers automatic encryption and key exchange for private and group messages. This can be controlled on a per-message basis with the `encrypt` arguments to *goTenna.driver.Driver.send_private()* and *goTenna.driver.Driver.send_group()*. If this parameter is `True`, the message will be encrypted.

The SDK will automatically exchange public keys with other private GIDs to which private messages are sent, and group invitation messages carry with them the public key for the group. Key exchange requires sending and receiving one additional message. Key exchange must be completed before an encrypted message can be sent to a private GID which has not been communicated with before, and so the first message to a given private GID may take additional time.

Though encryption can be disabled for messages sent from the SDK, the SDK will always respond to key exchange requests received over the goTenna network from other devices, and will always try to decrypt incoming encrypted messages.

The goTenna SDK features a storage interface definition in the form of `goTenna.storage.StorageInterface`, which can be subclassed to provide a storage implementation that meshes well with the rest of your application. The SDK also provides a default implementation, `goTenna.storage.EncryptedFileStorage`, which stores data in a file (by default `./.goTenna`) that is encrypted with your SDK token.

The SDK will store - The private key generated for the private GID with which you configured the SDK - Public keys for any other private GIDs with which key exchange has been completed - Groups the configured private GID has joined

A storage instance is passed in to the driver when it is initialized. If nothing is passed, the driver will use a `goTenna.storage.EncryptedFileStorage`.

2.7 Creating a Driver and Calling Methods

Though the example application has a lot of code, most of it is dedicated to parsing arguments from strings. If data to be sent and received are coming from elsewhere in the application, use of the goTenna SDK is very simple.

This is an example of creating an SDK thread and sending a broadcast message:

```python
import goTenna

# Put your SDK token in here; the program won't work and in fact won't
# be valid Python until you do
SDK_TOKEN =
# Because this function will be accessed by both the main thread
# (in the test_api function) and the SDK thread (in the event callback)
# we have to interact with it through object references only and never change
# its value. We could also use a queue here.
connected = [False]

# This event callback will let us know when we connect or disconnect
def event_callback(event):
    if event.event_type == goTenna.driver.Event.CONNECT:
        connected[0] = True
    elif event.event_type == goTenna.driver.Event.DISCONNECT:
        connected[0] = False

def test_api(gid): # this method should be passed the owner's GID
    # Create some RF settings. The example here uses frequencies that would not
    # work with a real device; substitute them with frequencies that you are allowed
    # to use.
    bandwidth = goTenna.constants.BANDWIDTH_KHZ[2]
    power = goTenna.constants.POWERLEVELS.HALF_W
    rf_settings = goTenna.settings.RFSettings(control_freqs=[150000000],
                                              data_freqs=[151000000],
                                              bandwidth=bandwidth,
                                              power_enum=power)
    geo_settings = goTenna.settings.GeoSettings(region=1)
    settings = goTenna.settings.GoTennaSettings(rf_settings,    # for goTenna PRO
                                                geo_settings)  # for goTenna Mesh
    # Create the driver. You can specify RF settings here, or wait; you can also
```

(continues on next page)

(continued from previous page)

```
    # reset them later
    api = goTenna.driver.Driver(SDK_TOKEN, gid, # Our GID and SDK token and default␣
↪settings
                                settings,
                                event_callback)
    api.start()
    while not connected[0]:
        pass
    api.send_broadcast("Hello, world!")

test_api(goTenna.setting.GID.generate(goTenna.settings.GID.PRIVATE))
```

2.8 Incoming Messages

The *goTenna.driver.Driver* instance is always polling the device while it is connected. It polls for both device status information (for instance, battery level) and new messages. When a new message arrives, the driver pulls it from the connected goTenna and either

- Discards it, if the message's destination is not one of the GIDs configured by the application. This happens if, for instance, the goTenna was configured once with GID `123456789098765`, then disconnected but left on, and received private messages. Then it is connected and configured with, say, GID `22222222222222`. While it correctly received the message for `123456789098765` (because at the time, that was its private GID) we will not present that message to an application that has configured with a different GID.
- Deals with it internally, if the message is not a user-facing type. For instance, group creation messages are not presented to the user; the SDK handles them internally, and then presents a *goTenna.driver.Event.GROUP_CREATE* event to the application.
- Presents it to the application, through the event callback with a *goTenna.driver.Event.MESSAGE* event.

2.9 Groups

Groups are associations of between three and twelve private GIDs, a group GID to identify the group, and a shared secret used for encryption. These groups are stored by all participants. When a group is created, invitations are sent to all identified members.

There are two distinct scenarios to group creation:

1. You create a group, that has not previously existed, with a list of members. You create a *goTenna.settings.Group* instance with the *goTenna.settings.Group.create_new()* factory method and your list of desired members. You invoke *goTenna.driver.Driver.add_group()* with that group, specifying `invite=True` and an `invite_callback`. The SDK generates and sends group invitation methods, calling your `invite_callback` when each of them is acknowledged, and finally calls the `method_callback` when all invitations have either been acknowledged or timed out and the group GID has been added to your connected device. You can now send and receive group messages with that group GID. You should save the group in the application for whenever you want to reconnect to the same device, but no further configuration is immediately necessary.
2. You receive an invitation to a group. The group GID is added to your goTenna. Your `event_callback` is invoked with the *goTenna.driver.Event.GROUP_CREATE* event and the *goTenna.driver.Event.group* attribute containing the *goTenna.settings.Group* object representing the group. You can now send and receive messages with that group. You should save the group in the application for whenever you want to reconnect to the same device, but no further configuration is immediately necessary.

CHAPTER

THREE

USING THE SAMPLE APP

The goTenna SDK comes bundled with a Python sample application giving an example of how to use the goTenna SDK. If you received the SDK from goTenna, the sample app is in the SDK zip as `sample.py`.

The sample app is a program that reads and executes commands from the command line and sends responses from the SDK back to the command line. Once you have installed the goTenna SDK, you can run the sample app by invoking `python sample.py`.

You should be presented with a welcome message and the command prompt `goTenna>`. The first thing to do is enter your SDK token with the `sdk_token` command. For instance, if your SDK token was `gotennasdk`, the command would be `sdk_token gotennasdk`.

Once the token is loaded, entering `help` or `?` will display a list of commands, and entering `help COMMAND` for a given command - for instance, `help send_broadcast` - will give you usage information on that command.

Responses from the SDK - for instance, notifications or incoming messages - are displayed on the same command line, and typing will type over those notifications.

Detailed documentation about available commands is below; however, to get started as quickly as possible you can do the following:

1. After installing the goTenna SDK, run the sample app: `python sample.py`
2. Enter your SDK token (this was distributed with the SDK) at the prompt: `sdk_token MY_TOKEN`
3. Plug in a goTenna, and turn it on. You should see a message like : `goTenna physically connected, configure to send messages.`
4. Configure a local GID: `set_gid 12345678909876` (or any other number; this is the goTenna equivalent of a phone number, and should be around 15 digits.)
5. For a goTenna PRO:
    a. Configure frequency settings: `set_frequencies 150000000 151000000` specifying one control frequency and one or more data frequencies in Hz.
    b. Configure a transmit power: `set_transmit_power HALF_W` (or `ONE_W`, `TWO_W`, or `FIVE_W`
    c. Configure your channel bandwidth in kHz: `set_bandwidth 11.8` (as specified in `list_bandwidth`)
6. For a goTenna MESH:
    a. Configure the geo region: `set_geo_region 1` (as specified in `list_geo_region`)
7. Watch for the message `Connected!`
8. Send a broadcast message on your goTenna: `send_broadcast Hello, world!`

3.1 Configuration

A goTenna (and the sample app) requires certain configuration options to operate: one private GID that identifies it on the network, and settings that tell it how to communicate. The private GID is set with the `set_gid` command:

```
set_gid GID
```

where `GID` is a most often 14 digit number. If you specify an invalid GID, the system will print an error message; run the command again with a valid GID.

A goTenna PRO requires Radio settings. Radio settings are configured in multiple parts. The first is the frequency settings:

```
set_frequencies CONTROL_FREQ DATA_FREQS
```

You must specify one control frequency and up to 16 data frequencies. These are specified in Hz, and must be in the 142-175 MHz or 445-480 MHz bands.

The second radio setting is the transmit power:

```
set_transmit_power POWER
```

We don't allow setting arbitrary output powers; you must pick `HALF_W`, `ONE_W`, `TWO_W`, or `FIVE_W`.

The third radio setting is the channel bandwidth:

```
list_bandwidth
```

```
set_bandwidth BANDWIDTH
```

The list command will display a list of valid bandwidths.

A goTenna MESH requires a Geo region setting. A Geo region is configured with the region number corresponding to its geo location:

```
list_geo_region
```

```
set_geo_region REGION
```

The list command will display a list of valid regions, that can then be specified with the set command.

3.2 Messaging

Once the GID and RF settings are configured, a device can connect. Once a device is connected, you can send messages from it and receive messages on it.

To send a broadcast, use the `send_broadcast MESSAGE` command. `MESSAGE` is the message you want to send. The transmission is successful when the sample app prints the phrase `Method call send_broadcast <your message> succeeded.`.

To send a group message, use the `send_group GROUP_GID MESSAGE` command. `GROUP_GID` should be the GID of the group to send the message to. `MESSAGE` can be an arbitrary sequence of text. Like broadcast messages, group messages print a phrase when they send successfully.

To send a private message, use the `send_private GID MESSAGE` command. `GID` should be the GID to send the message to. `MESSAGE` can be an arbitrary sequence of text. Like broadcast and group messages, private messages will print a success message; in addition, private messages are acknowledged by the recipient, and if the recipient receives and acknowledges the message the sample app will print a message like `Private message to <destination>: delivery confirmed`.

3.3 Groups

Message groups are sets of goTennas that are informed about a common GID, called a group GID, and the GIDs of the other members of that group. Groups can be created by calling `create_group`:

```
create_group MEMBER_GIDS...
```

You must specify the other members of the group by their private GIDs. When `create_group` is called, the sample app will both inform your connected goTenna about the newly-created group, and send private messages to the members you specified informing them about the group. If those goTennas are activated and on the network, they will acknowledge the invitations and the sample app will print `Invitation of GID to GROUP succeeded`. If any are disconnected from the network, the sample app will print `Invitation of GID to GROUP succeeded`. If an invitation failed, you can resend the invitation at a later time. Any goTenna that responded positively to the invitation will see any subsequent messages sent to that group GID.

`create_group` will also generate a new group GID, and print it. This group GID is the GID to specify when using `send_group`. The newly-created group is stored in the secure storage used by the SDK and will be immediately available for use, as well as loaded the next time you start the sample app with the same SDK token and private GID.

Once the connected goTenna has been informed of the new GID and all invitations have either been acknowledged or timed out, the sample app will print `Group GID created` or `Group GID could not be created`. `GID` will be the GID of the group.

> **note** In general, invited goTennas not acknowledging a group invitation is not a failure state for group creation since those invitations can be resent. We consider the invitation failed if and only if either the remote goTenna would not accept the new group GID, or precisely 0 invitations were acknowledged.

Group invitations may be resent later by calling `resend_invite`:

```
resend_invite GROUP_GID MEMBER_GID
```

Here you specify the GID of the group previously created by `create_group` and the member to invite.

> **note** Because members can not be added to or removed from groups once the group is created, groups are kept around for the lifetime of the sample app. Groups are created with `create_group` or by receiving a group invitation, and are stored locally. `resend_invite` checks that both the specified group is known, and the specified member GID is in the group already.

To send a message to a group, use `send_group`:

```
send_group GROUP_GID MESSAGE
```

This sends the specified message to the specified group.

3.4 Encryption

The sample app enables encryption for all group and private messages. This is controlled by the `_do_encryption = True` line in the sample app. If you change this to `False`, messages will no longer be encrypted, except for group invitations.

3.5 Storage

The sample app configures the SDK to use an encrypted file in its directory, `.goTenna`, as storage. This file is encrypted using your SDK token and, for each private GID you use with the sample app, contains

- The private key for your GID

- The public keys for any other goTennas with which you have exchanged encrypted messages
- Any groups that you have joined, including the group key and the other members of the group

3.6 Getting Messages And Events

The sample app's event handler prints connection, disconnection, incoming message events, and group invitations to the terminal.

CHAPTER

FOUR

GOTENNA PACKAGE

4.1 Package Contents

Most of the goTenna SDK is contained in the various submodules of the goTenna package. The most important submodule, and the primary method of SDK interaction, is *goTenna.driver*.

In addition to the submodules, the goTenna SDK package has the members `goTenna.api_version` and `goTenna.__version__` containing dotted version strings.

4.2 Submodules

4.3 goTenna.driver module

The primary module for managing the goTenna SDK.

This module provides classes and methods for general goTenna use. Instantiate the *Driver* class once in the application, and call `Driver.start()`, which is inherited from `threading.Thread`, to begin its background thread handling. It will then automatically scan for and connect to a goTenna device.

class goTenna.driver.**Driver**(*sdk_token*, *gid*, *settings*, *event_callback*, *poll_frequency=None*, *do_automatic_connect=True*, *device_blacklist=()*, *device_whitelist=()*, *device_types=()*, *shortname="*, *storage=None*)

Bases: `goTenna.driver_backend.DriverBackend`

A thread that will run SDK communications. This allows SDK consumers to delegate time-consuming and blocking tasks to the SDK.

This thread should be kept alive for the duration of the use of the SDK.

The intended use of this class is as the primary driver for the SDK. It spins up its own thread in the background, and communicates with the rest of the program through events. There is an event callback specified with the *Driver* object, which is used to communicate system level-events, for instance device connection or disconnection.

event_handler(*event*)

A callback called by the SDK driver whenever an event occurs.

Parameters **event** (`goTenna.driver.Event`) – The event that has occurred. The event type is denoted by the *goTenna.driver.Event.event_type* attribute, and the other members of the event contain additional data. These events can be used by the rest of the application to make decisions, for instance showing on the UI whether a device is connected or not.

Note This callback is called in the context of the SDK thread. If it communicates with the rest of the program in the main thread it should do so in a cross-thread way, for instance with a `Queue` or similar. This also means that the *`Driver.join()`* method that this class inherits from `threading.Thread` may not be called from this callback, or the base class will raise a `exceptions.RuntimeError`. If you want to quit from the event callback, call *`goTenna.driver.Driver.stop()`*, set some event, and then call *`Driver.join()`* from the main thread.

Methods called on the driver typically take a callback function that is invoked when the method succeeds or fails. This is to prevent the application from blocking execution while a command that can take some time - for instance, a message in a very congested environment - executes. Methods of this type, for instance *`goTenna.driver.Driver.send_private()`* take an argument called `callback`:

`callback`(*correlation_id*, *success=None*, *results=None*, *error=None*, *details=None*)
A callback called when a method ends.

Note: The arguments to this function are internally passed via keyword arguments. The function prototypes should have the exact argument names listed here.

Parameters

- **`correlation_id`** (*`uuid.Uuid`*) – A given method will return a UUID immediately when it invokes a command on the remote device. When the command ends and the callback is invoked, the same UUID is passed as the first argument to the callback. This enables the use of a generic successcallback function that still knows which method it is referring to.
- **`success`** (*`bool`*) – `True` if the method call succeeded; `None` if it failed.
- **`results`** – If the method returns a result, and it succeeded, the result will be here. The type depends on the method (and methods returning results will document the type).
- **`error`** (*`bool`*) – `True` if the method call failed; `None` if it succeeded.
- **`details`** (*`dict`*) – If the method failed, the `details` argument will contain a dict with more information on the failure. It will have a `code` key containing a member of *`goTenna.constants.ErrorCodes`*, and a `message` key containing a longer message.

So, calling methods on the driver is done this way:

```
in_flight = set()
def method_callback(correlation_id,
                    success=None, results=None,
                    error=None, details=None):
    if success:
        print("Message CORR_ID %s succeeded!" % str(correlation_id))
    elif error:
        print("Message CORR_ID %s failed: %s: %s" % (str(correlation_id),
                                                      str(details['code']),
                                                      str(details['msg'])))
    in_flight.pop(correlation_id)

in_flight.add(driver.send_broadcast("Hello, world!"))

# If we want, we can wait until the message is delivered by querying in_flight
# But we don't have to if we want to return to, for instance, GUI handling
```

The method may do local error checking (for instance, checking whether the driver is connected, or verifying an argument's type) before sending the message; if these checks fail, it will raise an exception. Which exception is raised is documented on a per-method basis.

Using the driver is as easy as creating the event callback, selecting a GID, and then creating and running the SDK Driver class:

```
status = [{}]
# This callback will be called from the SDK thread
def event_handler(event):
    if event.event_type == goTenna.driver.Event.CONNECT:
        print("Connected to %s" % event.device_details['port'])
    elif event.event_type == goTenna.driver.Event.DISCONNECT:
        print("Disconnected!")
    elif event.event_type == goTenna.driver.Event.MESSAGE:
        print("%s received from %s: %s" % (str(event.message_type),
                                           str(event.sender),
                                           str(event.msssage)))
    elif event.event_type == goTenna.driver.Event.STATUS:
        status[0] = event.status

# Use your pre-distributed SDK token here
driver = goTenna.driver.Driver(SDK_TOKEN, None, None, event_handler)

# We can set our local GID, assuming it's stored in a ``gids`` variable
driver.set_gid(gids[0])

# And some default RF settings - these will have to be changed to your
# settings before the code will work
bandwidth = goTenna.constants.BANDWIDTH_KHZ[2]
settings = goTenna.settings.RFSettings(control_freqs=[140000000],
                                       data_freqs=[141000000],
                                       bandwidth=bandwidth)
driver.set_rf_settings(goTenna.settings.RFSettings.from_defaults())

# Finally, start the driver
driver.start()
```

Finally, to call methods on the SDK thread, call methods of the driver.

Note Messages sent with this driver are ratelimited. Though they can be enqueued through calls to *send_broadcast()* or *send_private()* as fast the caller desires, they will always be executed such that there is a minimum one second delay between the completion of one message and the beginning of another.

__init__(*sdk_token*, *gid*, *settings*, *event_callback*, *poll_frequency=None*, *do_automatic_connect=True*, *device_blacklist=()*, *device_whitelist=()*, *device_types=()*, *shortname=''*, *storage=None*)

Build a *Driver* object.

Parameters

- **sdk_token** (*str*) – The SDK access token distributed along with this SDK. This is a fairly long string (the exact length and contents vary) identifying the developer.
- **gid** (*goTenna.settings.GID* or None) – The GID identifying the user. If None, it may be specified later with *set_gid()*. However, while a GID is not set, no device will be connected.
- **settings** – goTenna.settings.Settings object to configure connected goTen-

nas with. If `None`, no settings will be configured. While the settings are not set, no device will be connected.

- **`event_callback`** (*`func(event_type)`*) – A callback that will be called when new data is available for either device status or incoming messages. This will be called in the context of the SDK thread.
- **`poll_frequency`** (*`datetime.timedelta`*) – How frequently to poll the device for connection status and new messages. Default: 2 seconds.
- **`do_automatic_connect`** (*`bool`*) – Automatically attempt to connect to present devices. To configure once the driver is created, use *`allow_automatic_connection()`*; to query, use *`will_connect_automatically`*
- **`or byteslike] device_blacklist`** (*`iterable[str`*) – An iterable of device serials to ignore. To configure after the driver is created, use `blacklist_device()` and `unblacklist_device()`; to query, use *`blacklisted_devices`*. This (or `device_whitelist`) can be used to tightly control which devices the *`Driver`* instance will connect to. If both blacklist and whitelist are specified, the whitelist takes precedence and the blacklist is ignored. By default, or if specified as a falsy value, this is empty.
- **`or byteslike] device_whitelist`** (*`iterable[str`*) – An iterable of device serials to limit connections to. To configure after the driver is created, use `whitelist_device()` and `unwhitelist_device()`; to query, use *`whitelisted_devices`*. This (or `device_blacklist`) can be used to tightly control which devices the *`Driver`* instance will connect to. If both blacklist and whitelist are specified, the whitelist takes precedence and the blacklist is ignored. By default, or if specified as a falsy value, this is empty.
- **`or byteslike] device_types`** (*`iterable[str`*) – A set of `DEVICE_TYPE` the driver is allowed to connect to. By default, or if specified as a falsy value, this is both device types. On Windows this is ignored and the driver will connect to any connected goTenna device.
- **`storage`** (`goTenna.storage.StorageInterface`) – An initialized storage object to call that descends from *`goTenna.storage.StorageInterface`* and implements it, or `None`. If `None`, *`goTenna.storage.EncryptedFileStorage`* will be used.
- **`shortname`** (*`str`*) – A short (less than 9 bytes encoded) name to identify the owner of the running driver. This name is encoded in sent messages.

Raises `exceptions.ValueError` – If the configured settings are invalid, or if the SDK token is invalid.

`accepted_device_types`
The `set` of currently accepted device types.

Add or remove members of `DEVICE_TYPES` to this `set` to change the kinds of device this *`Driver`* instance will connect to.

`add_group`(*group*, *method_callback*, *invite*, *invite_callback=None*)
Add a group to the connected goTenna.

Parameters

- **`group`** (`goTenna.settings.Group`) – The group to add.
- **`method_callback`** – Method callback for the overall group addition process.

- **invite** (*bool*) – If True, send group invitations for all members of the group other than member 0. This invokes *invite_to_group()* on each group member. If False, do not send invites. This is useful for informing the goTenna of a previously-created group.
- **invite_callback** – Mandatory if invite is True, ignored otherwise. A callback that will be invoked when each group member is either successfully invited or the invite fails.

The form for the invite_callback is:

invite_callback(*correlation_id*, *member_index*, *success=None*, *error=None*, *details=None*)
Invoked when a group invitation succeeds or fails.

Parameters

- **correlation_id** (*uuid.uuid4*) – The same correlation id returned by this function.
- **member_index** (*int*) – The member index within the group of the member that has responded to its invitiation.
- **success** (*bool*) – True if the invitation succeeded; False or None otherwise.
- **error** (*bool*) – True if the invitation failed; False or None otherwise.
- **details** (*dict*) – Contains a dict with a code and msg element, as in the method callback, specifying details about the failure.

If invite is true, each member of the group will be sent a private message informing them of the group's creation. When that private message is done, the invite_callback will be invoked with the group's GID, the individual's GID, and the status of the invitation.

If the invitation failed, it may be retried by the application at a later date.

If the invitation succeeded, sending it again will have no effect.

The group creation is considered done when

- The group GID has been sent to the connected device (a failure here means the creation will fail)
- One of the following two cases:
  - Invites were not requested (this cannot fail)
  - All invites either succeeded or failed. Note: The group creation is only considered to fail if _none_ of the invites succeeded. If even a single invitation succeeded, the group creation succeeds.

To resend an invitation, use *invite_to_group()*.

Note Because the invitation process consists of sending a private message to each member of the group, this method may take some time.

allow_automatic_connection(*new_state*)
Toggle whether the driver will automatically try to connect to devices.

If automatic connection is disabled, device_present event notifications will still be issued but the driver will not attempt to connect to a device unless and until the *connect()* method is called.

If a device is already connected when this method is called it will remain connected. To disconnect, use *disconnect()*.

blacklisted_devices
The set of currently blacklisted device paths.

Add or remove serials to this set to change the *Driver* instances blacklist.

can_connect
Check if the object has all the configuration required to connect to a goTenna device.

Now a synonym for *can_connect_to()* with device_type=None

can_connect_to(*device_type=None*)

Check if the object can connect to the types of devices it is configured to connect to.

Parameters **device_type** – One of `Driver.DEVICE_TYPES` or unspecified

If the device type is unspecified, check all device types that the object would connect to. If specified, check only that type of device.

To connect, the `Driver` instance must have a valid GID and either preconfigured groups or no groups.

In addition, to connect to a goTenna Pro the `Drvier` instance must have valid RF settings.

Returns bool Whether or not the object can connect to the device.

connect(*path*, *force=False*)

Connect to a device.

Parameters

- **or byteslike path** (*str*) – The path to connect to. This should be a present device, ideally one of the devices specified in a `device_present` event.
- **force** (*bool*) – Force a connection to the device. This will cause the driver to attempt to connect to a device even if it sees a lockfile left by another driver instance for the path. This does not have an OS level effect, so if the other instance is still connected to the device the connection will fail, but it can be used to clean up if a driver crashed and left a lockfile around (which should be rare).

This method must be used on a device that is already present, i.e. that exists in *devices_present*.

Raises

- **ValueError** – if the path does not exist
- **TypeError** – If an argument is the wrong type
- **RuntimeError** – If the driver is already connected

connected

`True` if the driver is currently connected to a device; `False` otherwise.

device_info

A dict containing information about the currently connected device. Has the keys

- `fw_version` (updated on connection, a tuple of (`major`, `minor`, `bugfix`))
- `serial` (updated on connection, a string)
- `bluetooth_enabled` (in this SDK version, always `False` because bluetooth is always disabled when a device connects, and reenabled before disconnecting)
- `battery_percentage` Updated every time the device is polled; the charge level of the battery as a percentage.
- `temperature` Updated every time the device is polled; the temperature of the system as a whole, in Celsius. Note that this temperature is not related to thermal throttling; it simply gives an overview of how hot the device is, as a whole.

device_type

The type of device the driver is currently connected on, or `None`.

If connected, this is one of `DEVICE_TYPES`.

devices_present
Return a set of physically connected goTenna devices suitable for passing to `force_connect_device()` or `blacklist_device()`.

The elements of the set are tuples of (product, serial_number, path). The serial number element is a `str` suitable for use with *whitelisted_devices* and *blacklisted_devices* The `product` element is one of `DEVICE_TYPES` The `path` element is a string indicating the path at which the device is present, suitable for use with *connect()*

disconnect()
Disconnect the currently-connected device.

Parameters callback (*callable*) – A method callback. Optional because the disconnect will incur a `disconnect` event anyway.

Raises

- **RuntimeError** – If no device is currently connected
- **TypeError** – If the callback is specified and not callable

echo(*callback*)
Flash the LED.

Parameters callback – The callback to call when the method completes (see *goTenna.driver.Driver*)

returns uuid.UUID Unique identifier for this method invocation.

Raises exceptions.RuntimeError – If no device is currently connected.

Note If using the SDK through this class, it is not necessary to use the `echo` command to diagnose connection status. Because the driver is constantly querying the device for status updates, any disconnection will be noticed within a couple seconds and the event callback will be invoked with a disconnect event.

geo_settings
Property for querying the configured GEO settings

gid
Property for querying the gid

groups
Property for querying configured groups

invite_to_group(*group*, *member_idx*, *method_callback*, *ack_callback=None*)
Resend a group invite.

Parameters

- **group** (goTenna.settings.Group) – The group in question. The currently-set private GID must be member 0 of this group, indicating that the local user created it.
- **member_idx** (*int*) – The index of the member withing `group.members` to re-invite.
- **method_callback** – The callback to call when the method completes (see *goTenna.driver.Driver*)
- **ack_callback** – The callback to call when an acknowledgement for the invite is received from the goTenna. This callback should take only a correlation ID `uuid.uuid`. For more information see *send_private()*.

Note This method cannot be used to add a new member to an already-created group; this is not a supported group interaction. To change group membership, create a new group.

join(*timeout=None*)

An override of `threading.Thread.join()` to quickly stop and dispose of the SDK thread.

Note This may not be called from the context of the SDK thread, including in an event or method callback. To stop the driver in those circumstances, call *`stop()`* and later, in the main thread, call *`join()`*.

next_counter

Utility to provide an incrementing counter for payloads.

port

The port the driver is currently connected on, or `None`.

If connected, this is a string encoding the platform-specific identifier for the connection (e.g. a COM port on Windows or a path in `/dev/` on Posix)

remove_group(*group, method_callback*)

Remove a group from the goTenna device.

This will remove the group from the device and remove it from the known groups (what is returned from *`groups`*). Messages received intended for this group will be discarded.

Removing a group is a local action only and is not broadcast over the network. Others with the group will still have access to it.

Parameters

- **group** (`goTenna.settings.Group`) – The group to remove.
- **method_callback** – The callback to call when the method completes (see *`goTenna.driver.Driver`*)

Raises

- **`TypeError`** – If an argument is the wrong type
- **`ValueError`** – If the group is not known (not in *`groups`*)

rf_settings

Property for querying the configured RF settings

run()

The override of `threading.Thread.run()` that is the driver thread's main method.

Should not be called directly - users should call `Driver.start()`, which is inherited from `threading.Thread`.

send_broadcast(*payload, callback, is_repeated=False*)

Send a broadcast message.

Note This message will be received by all devices within range tuned to the same frequency settings.

Parameters

- **payload** (`goTenna.payload.Payload`) – The message to send. *`goTenna.payload.Payload.valid()`* must be `True`.
- **callback** – The callback to call when the method completes (see *`goTenna.driver.Driver`*)
- **is_repeated** – Pass True if this message is one instance of a message that is frequently repeated - for instance, automatically scheduled location updates.

Returns uuid.UUID Unique identifier for this method invocation.

Raises

- **`exceptions.RuntimeError`** – If no device is currently connected.
- **`exceptions.TypeError`** – If the type of either message or callback is incorrect
- **`exceptions.ValueError`** – If the message is too long or the payload is not valid

`send_emergency`(*payload*, *callback*)
Send a emergency message.

Note This message will be received by all devices within range tuned to the same frequency settings.

Parameters

- **`payload`** (`goTenna.payload.Payload`) – The message to send. *`goTenna.payload.Payload.valid()`* must be `True`.
- **`callback`** – The callback to call when the method completes (see *`goTenna.driver.Driver`*)

Returns uuid.UUID Unique identifier for this method invocation.

Raises

- **`exceptions.RuntimeError`** – If no device is currently connected.
- **`exceptions.TypeError`** – If the type of either message or callback is incorrect
- **`exceptions.ValueError`** – If the message is too long or the payload is not valid

`send_group`(*group*, *payload*, *callback*, *encrypt=True*, *is_repeated=False*)
Send a message to a group.

Parameters

- **`group`** (`goTenna.settings.Group`) – The destination group. The currently set private GID must be a member of this group; if it is not, `ValueError` is raised.
- **`payload`** (`goTenna.payload.Payload`) – The payload to send.
- **`callback`** – The method callback (see *`goTenna.driver.Driver`*).
- **`encrypt`** (*`bool`*) – Encrypt the group message.
- **`is_repeated`** – Pass True if this message is one instance of a message that is frequently repeated - for instance, automatically scheduled location updates.

Note Like broadcast messages, group messages do not wait for acknowledgement. This method will call its callback as soon as it is sent.

Raises

- **`exceptions.RuntimeError`** – If no device is connected
- **`exceptions.ValueError`** – If the currently-set GID is not in the specified group or if the payload is not valid.
- **`exceptions.TypeError`** – If an argument's type is incorrect.

`send_private`(*other*, *payload*, *callback*, *ack_callback=None*, *encrypt=True*, *is_repeated=False*)
Send a private message.

Parameters

- **`other`** (`goTenna.settings.GID`) – The destination of the method.

- **payload** (`goTenna.payload.Payload`) – The payload to send. *`goTenna.payload.Payload.valid()`* must be `True`.
- **callback** – The callback to call when the method completes (see *`goTenna.driver.Driver`*)
- **ack_callback** (`callable` or `None`) – The callback to call when the private message is acknowledged by the receiver.

The form for the ack callback is

ack_callback(*correlation_id*, *success*)
A function called by the driver when a message acknowledgement is received.

Note: The system calls the ack callback with keyword arguments, so the function's arguments should be named exactly as described here.

Parameters
- **correlation_id** (*`uuid.uuid`*) – The same correlation ID returned by the call to *`send_private()`* and passed in to the method callback.
- **success** (*`bool`*) – Whether the acknowledgement succeeded (an acknowledge message was received from the network) or failed (timed out).

The callback is generated by receiving an acknowledgement message from the destination of the private message, over the goTenna network. If the destination is not on the network, for instance if it ran out of battery or moved too far away, the acknowledgement message may not occur. The system has built in timeouts that depend on the configured system bitrate that generate a negative acknowledgement in this case (the callback will be invoked with `success=False`).

If the callback is not specified, the incoming acknowledgement message will be discarded, and a warning logged.

Parameters

- **encrypt** (*`bool`*) – Encrypt the private message. If this is the first private message to `other`, the SDK will exchange keys with `other` automatically. This may incur additional messaging latency. However, during this latency more messages to `other` (or other destinations) may be enqueued as normal.
- **is_repeated** – Pass True if this message is one instance of a message that is frequently repeated - for instance, automatically scheduled location updates.

Raises

- **exceptions.RemoteError** – If no device is connected
- **exceptions.ValueError** – If the message was too long or the payload is invalid.

Returns uuid.UUID Unique identifier for this method invocation

set_emergency_beacon(*enabled*, *callback*)
Enable or disable the emergency beacon

If the emergency beacon is enabled, the emergency message configured with `set_emergency_method()` will be periodically broadcast. Once the beacon is enabled, further calls to *`set_emergency_message()`* will change the message that is being broadcast - for instance, to update the GPS location.

Note If the emergency beacon is enabled when no message is set, nothing will be broadcast until the message is set with *`set_emergency_message()`*. When the message is set, it will immediately begin broadcasting.

Parameters

- **`enabled`** (*`bool`*) – Whether to enable (`True`) or disable (`False`) the emergency beacon
- **`callback`** – The callback to call when the method completes and the beacon is enabled or disabled (see *`goTenna.driver.Driver`*).

Returns uuid.UUID Unique identifier for this method invocation.

Raises

- **`exceptions.RuntimeError`** – If no device is currently connected
- **`exceptions.TypeError`** – If the type of either message or callback is incorrect

`set_emergency_message`(*payload*, *callback*)

Set the message that will be broadcast if the emergency beacon is activated.

This method stores the payload internally on the connected goTenna rather than sending the message immediately. When emergency beacon mode on the goTenna is activated, it will send the stored emergency message. If emergency beacon mode is already activated when this method is called, the next emergency message will be the message sent through this method.

A common use case of this method is to frequently update the stored emergency message with current GPS coordinates.

Parameters

- **`payload`** (`goTenna.payload.Payload`) – The message to send. *`goTenna.payload.Payload.valid()`* must be `True`.
- **`callback`** – The callback to call when the method completes and the message has been stored (see *`goTenna.driver.Driver`*)

Returns uuid.UUID Unique identifier for this method invocation.

Raises

- **`exceptions.RuntimeError`** – If no device is currently connected
- **`exceptions.TypeError`** – If the type of either message or callback is incorrect
- **`exceptions.ValueError`** – If the message is too long or the payload is not valid.

`set_geo_settings`(*geo_settings*)

Change the device's loaded GEO settings (and cache the new settings locally).

Parameters **`geo_settings`** (`goTenna.settings.GeoSettings`) – The settings to update.

Raises

- **`exceptions.ValueError`** – If geo_settings.valid is not true
- **`exceptions.TypeError`** – If geo_settings is not an instance of *`goTenna.settings.GeoSettings`*

Note Like with setting the GID, changing the GEO settings disconnects and reconnects any connected goTenna device.

`set_gid`(*gid*)

Set the GID associated with the current user.

Parameters **`gid`** (`goTenna.settings.GID`) – The GID to associate. If `None`, no device will connect.

Note When a new GID is set, any connected goTenna will disconnect and reconnect.

Note When a new GID is set, any previously-configured groups will be deleted.

`set_operation_mode`(*mode*, *callback*)

Turn the device on or off, or set it to a relay-only mode.

goTenna devices have several different operating modes.

A goTenna device is never truly "off" when it is plugged into USB and able to communicate. It remains in a limited mode in which it consumes less power and does not communicate on the goTenna network; however, some functions are still available, such as updating the device firmware. When a goTenna that is turned off is plugged into USB, it boots into this limited mode.

This method allows an SDK user to set the device mode. This method is idempotent, so setting the goTenna device to a given mode multiple times will have the same effect as doing it once. It is therefore advisable to always set the device to the desired mode explicitly. The SDK does not automatically turn the goTenna device on upon connection in case the SDK use case involves the device continuing to appear off, for instance in a charging station.

Parameters

- **`mode`** (*`str`*) – 'normal', 'off', or (on goTenna 900 devices) 'relay'.
- **`callback`** – The callback to call when the method completes (see *`goTenna.driver.Driver`*)

Raises **`exceptions.KeyError`** – If `mode` is not a valid operation mode.

`set_rf_settings`(*rf_settings*)

Change the device's loaded RF settings (and cache the new settings locally).

Parameters **`rf_settings`** (`goTenna.settings.RFSettings`) – The settings to update.

Raises

- **`exceptions.ValueError`** – If rf_settings.valid is not true
- **`exceptions.TypeError`** – If rf_settings is not an instance of *`goTenna.settings.RFSettings`*

Note Like with setting the GID, changing the RF settings disconnects and reconnects any connected goTenna device.

`set_shortname`(*shortname*)

Set the driver short name.

The short name is sent along with messages.

Parameters **`shortname`** (*`str`*) – The shortname to use. Must be <9 bytes encoded.

`shortname`

The short name associated with this driver.

The short name is sent along with messages.

To change the short name use *`set_shortname()`*.

`stop`()

Safely stop the SDK thread, in preparation for quitting. This may be called from any context.

`storage`

The storage instance the `Driver` is configured with

`system_info`
Property for querying the most recent device information. Will return similar data to that specified in the status event, as well as some data that does not change frequently (for instance, the device serial).

`update_firmware` (*firmware_file*, *method_callback*, *progress_callback*, *version=None*)
Update the firmware of a connected goTenna device.

Parameters

- **`firmware_file`** (`str`) – The path to a firmware file on the system. If `version` is not specified, this method will attempt to pull the version from the filename by splitting it on . and parsing the first three results as *int*.
- **`version`** (`tuple[int, int, int] or None`) – The version of the firmware to upload. If `version` is not specified, this method will attempt to pull the version from the filename. If specified, it is `(major, minor, bugfix)`. If a value over 255 is specified for any field, it will be limited to the least significant byte.
- **`method_callback`** – The callback to call when the method completes (see `goTenna.driver.Driver`)
- **`progress_callback`** – A callback called by the system periodically to indicate the progress of the firmware update:

  `progress_callback` (*progress*, ***kwargs*)
  This callback is called periodically to update the SDK consumer on the progress of the firmware update. The progress indicated by this method may not linearly relate to the time left or elapsed in the update. When the progress indicates completion, the update may still not be done.

  The progress argument will be a float between 0 and 1. The further `kwargs` should be present for future compatibility.

Warning The progress callback should only be used for display purposes. The firmware update is only complete when the method callback is called

Note After the firmware update completes the device will be unresponsive for several tens of seconds, then disconnect. When it reconnects, it will have the new firmware.

Raises

- **`exceptions.RuntimeError`** – If no device is connected
- **`exceptions.ValueError`** – If `version` is not specified, and no version could be parsed from the filename; or, if the path specified in `firmware_file` does not exist.
- **`exceptions.TypeError`** – If an argument is of an invalid type

Returns uuid.UUID A UUID to correlate the method_callback invocation with the method invocation.

`whitelisted_devices`
The `set` of currently whitelisted device paths.

Add or remove serials to this `set` to change the `Driver` instances whitelist.

`will_connect_automatically`
Whether or not the driver will connect to a device on its own.

Change with `allow_automatic_connection()`.

class `goTenna.driver.`**`Event`** (*event_type*, ***kwargs*)
Bases: `object`

An object representing an event. This object is generated by the SDK and has attributes holding data about the event.

> **Note** Has a reasonable `__str__` method defined; if all you want to do is print the event, use `str(event)`

CONNECT = 0
A connect event

DEVICE_PRESENT = 5
A device-present event

DISCONNECT = 1
A disconnect event

GROUP_CREATE = 4
A group creation event

MESSAGE = 2
A message incoming event

STATUS = 3
A status update event

device_details = None
If this is a connection event, this attribute will hold a dict of device details: `fw_version`: A string holding the firmware version `serial`: The device serial number `port`: The name of the port where the device is connected (for instance, COM13 or /dev/ttyUSB0) `device_type`: The type of the device, one of `Driver.DEVICE_TYPES`

device_paths = None
If this is a device_present event, this will hold a list of dicts of the same format as *`device_details`* detailing the currently physically present devices.

disconnect_code = None
If this is a disconnect event, this will hold a member of *`goTenna.constants.ErrorCodes`* explaining the disconnect.

disconnect_reason = None
If this is a disconnect event, this will hold a human-readable string explaining the code.

event_type = None
The type of event. One of *`CONNECT`*, *`DISCONNECT`*, *`MESSAGE`*, *`STATUS`*, *`GROUP_CREATE`*, *`DEVICE_PRESENT`*

group = None
If this is a group creation event, this will hold the group object

message = None
If this is an incoming message event, this attribute will hold the message, an instance of *`goTenna.message.Message`*

status = None
If this is a status event, this attribute will hold the status dictionary:

- `"battery"`: The percentage of battery remaining
- `"temperature"`: The temperature of the device
- `"bluetooth_enabled"`: Whether Bluetooth is enabled or disabled

class goTenna.driver.**SpiDriver**(*bus_no*, *chip_no*, *req*, *rdy*, **args*, ***kwargs*)
Bases: *`goTenna.driver.Driver`*

A child of goTenna.driver.Driver that uses a SpiConnection to connect.

Use this class wherever you would previously use `goTenna.driver.Driver` to connect to a goTenna Pro Embedded.

`__init__`(*bus_no*, *chip_no*, *req*, *rdy*, **args*, ***kwargs*)

Build a SpiDriver.

Parameters

- **`bus_no`** (`int`) – The number of the spidev bus
- **`chip_no`** (`int`) – The chip select number of the device
- **`req`** – The request gpio pin, in any form that can be recognized by `GPIOMonitor.__init__()`
- **`rdy`** – The ready gpio pin, in any form that can be recognized by `GPIOMonitor.__init__()`

The remaining positional and keyword arguments are forwarded to `goTenna.driver.Driver.__init__`.

`stop`()

Stop the driver.

Ensures that the spi connection is stopped and delegates to `goTenna.driver.Driver.stop()`.

4.4 goTenna.settings module

Classes and functions for interacting with the settings

class `goTenna.settings.`**`GID`**(*gid_val*, *gid_type*, *via_gateway=None*)

Bases: `object`

A class containing the configuration of a contact. A contact has a GID, and some string representing a name. The string is used for cosmetic purposes and, if you desire, lookup.

Note: The SDK will usually store the numeric identifier part of the GID as a `int`. However, on python2 systems it is possible for `int` to be too short to contain the GID (it is 6 bytes, and python2 only guarantees 32 bites in `int`). If the SDK detects this is the case (by using `sys.max_int()`), GID numeric IDs will be stored as `long` and must be specified as `long`. If it is not the case, any arguments passed as `long` will be converted to `int` internally.

`PRIVATE = 0`

`GROUP = 1`

`BROADCAST = 2`

`EMERGENCY = 3`

`classmethod broadcast`()

Generate the broadcast GID.

This GID is used for all broadcast-class messages. It should not be used for identity.

`classmethod emergency`()

Generate the emergency GID.

This GID is used for all emergency-class messages. It should not be used for identity.

classmethod gateway()

Generate the gateway GID.

This GID is used as the private GID for all gateway devices. It should not be used for anything else.

classmethod generate(*gid_type*)

Generate a new GID that should be unique (through use of `uuid`)

Parameters **gid_type** – A GID type, either *PRIVATE* or *GROUP* (the broadcast messages have fixed GIDs and thus cannot be generated).

Raises **exception.KeyError** – if `gid_type` is invalid.

Return GID The newly-created GID.

static type_name(*gid_type*)

Get a human-readable name for a GID type.

Parameters **gid_type** (*int*) – One of *PRIVATE*, *GROUP*, or *BROADCAST*.

Returns str The name of the type.

Raises **exception.KeyError** – If `gid_type` is not one of the above.

static type_code(*type_name*)

Get a gid_type from a human readable name (i.e. as returned from *type_name()*)

Returns int The GID type, suitable for passing to *__init__()*.

Raises **exception.KeyError** – If `type_name` is not one of those returned by *type_name()*.

__init__(*gid_val*, *gid_type*, *via_gateway=None*)

Build an instance of *GID*.

Parameters

- **gid_val** – The GID. This should be an integral type in [0, *goTenna.constants.GID_MAX*]. On python2, `long` is always accepted and `int` is accepted if `sys.maxint()` is > 0xffffffffffff (6 bytes unsigned). The value is stored internally as `int` if possible (same criterion as accepting an `int` as input) and `long` otherwise. On python 3, only `int` is accepted and used as the backing store.
- **gid_type** (*int*) – The type of the GID. This should be one of *PRIVATE*, *GROUP*, *BROADCAST*, or *EMERGENCY*
- **via_gateway** (`goTenna.settings.GID` *or* `None`) – the GID of the gateway this ID is accessible through, if it is remote; otherwise `None`. If a gateway is specified, it should be the value returned by *goTenna.settings.GID.gateway()*.

Raises

- **exception.TypeError** – If `gid_val` is not an integral type.
- **exception.ValueError** – If `gid_val` is outside valid bounds.
- **exception.KeyError** – If `gid_type` is not valid.

gid_type

The type of the GID.

Returns int *GID.PRIVATE*, *GID.GROUP*, or *GID.BROADCAST*

gid_val

The numeric value of the GID.

Returns int The GID.

`via_gateway`
The gateway by which this GID is accessible, or `None` if it is local.

Returns GID or None The gateway

`to_dict``()`
Create a dict representing this class (e.g. for serialization)

`class` `goTenna.settings.`**`Group`**`(`*group_gid*, *members*, *shared_secret*`)`
Bases: `object`

A named tuple containing the configuration of a group. A group

- Is associated with a specific group GID, which identifies the group
- Is associated with a specific private GID, which is the local owner
- Contains the membership index of that owner GID
- Contains a list of other GIDs and membership index making up the rest of the group membership

Owner, member, and group GIDs are stored as *`GID`*

`MAX_MEMBERS = 11`

`classmethod from_dict``(`*settings_dict*`)`
Factory for building a Group object from a settings dict.

Parameters **`settings_dict`** (*`dict`*) – A dictionary of settings to scan.

`classmethod create_new``(`*members*`)`
Factory for creating a new Group object that does not yet exist on the network.

Parameters **`members`** (*`list`*`[goTenna.settings.GID]`) – The members of the group to create. This should include the owner of the group at index 0.

To create a new Group from a notification, use `from_invite()`.

`__init__``(`*group_gid*, *members*, *shared_secret*`)`
Build a Group object.

Parameters

- **`group_gid`** (`GID`) – The GID/name that identifies the group
- **`members`** (*`list`*`[GID]`) – Members of the group, including the local user. The member index will be generated from an individual's location in the list.
- **`shared_secret`** (*`None or bytearray`*) – The cryptographic shared secret. It should be a byteslike of 32 bytes precisely.

This method will generate group indices for all members. The local user should be specified along with the rest of the group members.

There must be at least two members.

Note This constructor should rarely be used directly; instead, use one of the factory methods `settings.Group.create_new()`, `settings.Group.from_dict()`, or `settings.Group.from_invite()`

Note Groups are considered to be immutable once created. Their members are accessed through read-only accessors. To change the membership of the group create, a new group and stop using the old one.

Raises

- **exception.ValueError** – If there are fewer than 3 or more than *MAX_MEMBERS* members, or if the shared secret is invalid
- **exception.TypeError** – If the type of one of the arguments is incorrect.

members

Read-only accessor for group members.

Note Returns a copy of the member list. Modifying the member list returned by this property will have no effect on the member list stored in the group.

shared_secret

Read-only accessor for the group shared secret.

Note Returns a copy of the shared secret. Modifying the shared secret returned by this property will have no effect on the secret stored in the group.

gid

Read-only accessor for the group GID.

Note Returns a copy of the GID. Modifying the GID returned by this property will have no effect on the GID stored in the group.

to_dict()

Return the settings that need to be stored in a dict.

class goTenna.settings.**RFSettings**(*data_freqs=None*, *power_enum=None*, *control_freqs=None*, *bandwidth=None*)

Bases: object

A class containing RF settings.

This defines control and data frequencies, transmission power, bandwidth, and datarate.

RFSettings objects must include at least one control frequency and at least one data frequency. All frequencies must be within the bands supported by goTenna.

The power and bandwidth values must be valid supported values.

It defines a to_dict() method for easy serialization.

__init__(*data_freqs=None*, *power_enum=None*, *control_freqs=None*, *bandwidth=None*)

Build an RFSettings object.

Though all the parameters in the constructor are optional, this is to allow early instantiation and late configuration of the object. An RFSettings object cannot be used to configure a device until it itself is fully configured. You can check if the object is fully configured by using the *RFSettings.valid* property.

Parameters

- **data_freqs** (*list[int]*) – An iterable containing frequencies of the data channels, in Hz
- **power_enum** (goTenna.constants.POWERLEVELS) – The transmission power. One of the values defined in goTenna.constants.POWERLEVELS
- **control_freqs** (*list[int]*) – The frequencies to use for control channels.
- **bandwidth** (goTenna.constants.Bandwidth) – The bandwidth for the channels specified here. This must be a member of *goTenna.constants.BANDWIDTH_KHZ*.

The settings here implicitly control the frequency-hopping behavior of the device. The device will move through the list of configured frequencies in the order specified by the lists owned by this object. To change the order of frequency hopping, change the order of the frequencies in the list.

There must be a minimum of one data frequency and one control frequency specified in this object. However, these need not be unique. To configure the device to only ever use a single frequency, specify the same frequency as the single control and data channel.

to_dict()
Return the settings that need to be stored in a dict.

static from_dict(*settings_dict*)
Helper for building an RFSettings class from a dict of settings

Parameters **settings_dict** (`dict`) – The dict to scan

valid
Determine whether the RFSettings object contains valid settings.

bandwidth_valid
Determine whether the bandwidth of the object is valid.

This is a subset of the validation performed by `goTenna.settings.RFSettings.valid()` and is used by that method.

freqs_valid
Determine whether the frequency content of the object is valid.

This is a subset of the validation performed by `goTenna.settings.RFSettings.valid()` and is used by that method.

static validate_freq(*freq*)
Validate whether a given frequency is in an acceptable band.

Parameters **freq** (`int`) – The frequency to check.

Return bool True if the frequency is acceptable.

Acceptable bands are defined by `goTenna.constants.BAND`

class goTenna.settings.**GeoSettings**(*region=None*)
Bases: `object`

A class containing Geo settings.

This defines the operating region. The region value must be valid supported value.

It defines a to_dict() method for easy serialization.

__init__(*region=None*)
Build an GeoSettings object.

Though all the parameters in the constructor are optional, this is to allow early instantiation and late configuration of the object. An GeoSettings object cannot be used to configure a device until it itself is fully configured. You can check if the object is fully configured by using the *GeoSettings.valid* property.

Parameters **region** (`goTenna.constants.GEO_REGION`) – The geo region. One of the values defined in goTenna.constants.GEO_REGION

to_dict()
Return the settings that need to be stored in a dict.

static from_dict(*settings_dict*)
Helper for building a GeoSettings class from a dict of settings

Parameters **settings_dict** (`dict`) – The dict to scan

valid
Determine whether the GeoSettings object contains valid settings.

class goTenna.settings.**SpiSettings**(*bus_no=0*, *chip_no=0*, *request_gpio=22*, *ready_gpio=27*)

Bases: `object`

A class containing SPI settings.

This defines the SPI settings, defaulted to allow a connection with a Rasp PI on /dev/spidev0.0

It defines a to_dict() method for easy serialization.

__init__(*bus_no=0*, *chip_no=0*, *request_gpio=22*, *ready_gpio=27*)

Build an SpiSettings object.

Though all the parameters in the constructor are optional, this is to allow early instantiation and late configuration of the object. An GeoSettings object cannot be used to configure a device until it itself is fully configured. You can check if the object is fully configured by using the *SpiSettings.valid* property.

to_dict()

Return the settings that need to be stored in a dict.

static from_dict(*settings_dict*)

Helper for building a SpiSettings class from a dict of settings

Parameters **settings_dict** (`dict`) – The dict to scan

valid

Determine whether the SpiSettings object contains valid settings.

class goTenna.settings.**GoTennaSettings**(*rf_settings=None*, *geo_settings=None*)

Bases: `object`

A class containing goTenna settings such as the RFSettings and GeoSettings.

__init__(*rf_settings=None*, *geo_settings=None*)

Build an GoTennaSettings object.

rf_settings_valid

Determine whether the RFSettings object contains valid settings.

geo_settings_valid

Determine whether the GeoSettings object contains valid settings.

4.5 goTenna.constants module

Module containing useful contents for communicating with goTennas.

goTenna.constants.**sharing_frequency**(*name*)

Look up a location sharing frequency from its name.

Parameters **name** (`str`) – A key (any case) of `SHARING_FREQUENCIES`

Returns int The code of the frequnecy

Raises **KeyError** – If `name` is not a valid sharing frequency

goTenna.constants.**sharing_frequency_name**(*frequency_code*)

Look up the name of a sharing frequency

Parameters **frequency_code** (`int`) – The code, a value of `SHARING_FREQUENCIES`

Returns str The key of the frequency

Raises **KeyError** – If `frequency_code` is not in `SHARING_FREQUENCIES` goTenna.constants.**MESSAGE_TYPES = {'broadcast': 2, 'emergency': 3, 'group': 1, 'private**

Specify the type of a message used with send-generic

goTenna.constants.**message_type**(*name*)

Look up a message type from its name.

Parameters **name** (*str*) – A key of *MESSAGE_TYPES*

Returns int The code of the type

Raises **KeyError** – If `name` is not a valid message type name

goTenna.constants.**message_type_name**(*type_code*)

Look up a message type name from a code

Parameters **type_code** (*int*) – A value of *MESSAGE_TYPES*

Returns str The name of the type code

Raises **KeyError** – If `type_code` is not a value of `constants.MESSAGE_TYPES`

class goTenna.constants.**POWERLEVELS**

Bases: `object`

A class defining acceptable values for transmit power. Only the values defined in this class should be stored in RFSettings.

HALF_W = 0

ONE_W = 1

TWO_W = 2

FIVE_W = 3

static valid(*value*)

A validator to ensure a given value is OK

static name(*value*)

Get the name of a power level from its value

static value(*name*)

Get the value of a powerlevel from a string name

goTenna.constants.**MINIMUMVERSION = (0, 14, 35)**

The minimum version this SDK will work with. Checked by *goTenna.driver.Driver* when it connects to a device; to manually check, use *goTenna.constants.version_ok()*

goTenna.constants.**MAXIMUMVERSION = (256, 256, 256)**

The maximum version this SDK will work with. Checked by *goTenna.driver.Driver* when it connects to a device; to manually check, use *goTenna.constants.version_ok()*

goTenna.constants.**version_below**(*version_a*, *version_b*)

Compare two version tuples.

Parameters

- **version_a** (*tuple(int, int, int)*) – The first version
- **version_b** (*tuple(int, int, int)*) – The second version

Returns bool True if version_a < version_b; False otherwise

goTenna.constants.**version_ok**(*version_major*, *version_minor*, *version_bugfix*)

Check if the version tuple (expanded into the arguments) from a firmware is OK with the stored acceptable version of the SDK.

Parameters

- **`version_major`** (*`int`*) – The major version of the firmware.
- **`version_minor`** (*`int`*) – The minor version of the firmware.
- **`version_bugfix`** (*`int`*) – The bugfix version of the firmware

Returns bool True if the firmware and SDK versions are compatible. False otherwise.

Note This method is called automatically on connection by *`goTenna.driver.Driver`*. Most use cases of the SDK do not require calling it directly.

class `goTenna.constants.`**`ErrorCodes`**

Bases: `object`

A class with members denoting remote error codes that the goTenna device may send. These are used as the `code` value of *`goTenna.constants.RemoteError`*

`NOMETHOD = -1`
No such method exists

`TIMEOUT = -2`
Time out executing command

`OSERROR = -3`
OS error communicating with device

`EXCEPTION = -4`
Internal Python exception

`VERSION = -5`
Version mismatch

`KEY_EXCHANGE_FAILED = -6`
Key exchange failed

`NONE = 0`
No error has occurred.

`UNKNOWN = 1`
An unknown error on the device has occurred.

`FLASH = 2`
An error has occurred within the device's long term storage handling.

`INVALID_DST = 3`
An error has occurred within the device's internal message passing.

`UNEXPECTED_CMD = 4`
An unhandled command was sent to the device.

`INVALID_DATA = 5`
Invalid arguments were passed to the device

`UNAVAILABLE = 6`
RF communication was unavailable or a message failed to send.

`RETRY_LATER = 7`
Current conditions make RF communication impossible, the command should be retried.

`SW_ERROR = 8`
Internal error.

RECEIVER_UNAVAILABLE = 9
The intended receiver is not currently available; the message may be retried when it is expected the receiver is present.

ACK_NOT_RECEIVED = 10
No acknowledgement was received for the current message.

RANDOM_BACKOFF_EXHAUSTED = 11
The device was unable to find a clear timeslice to broadcast in.

THERMAL_BACKOFF = 12
The device was too hot and needed to wait to retry

BUSY_CHANNEL = 13
The channel is currently busy.

KEY_EXCHANGE_COMPLETE = 14
The built in key exchange protocol with the specified destination is complete, and messages may now be sent.

KEY_EXCHANGE_PARTIAL = 15
Key exchange with the specified destination is ongoing. Messages should be sent until the key exchange is complete.

TRX_NOT_READY = 16
The RF section of the device is not ready to operate.

exception `goTenna.constants.`**TimeoutException**(*msg*, *timeout*)
Bases: `exceptions.OSError`

An exception representing a command that timed out waiting for its success message.

timeout = None
The timeout that was exceeded

msg = None
A message to help identify the command

exception `goTenna.constants.`**RemoteError**(*code*, *msg*)
Bases: `exceptions.Exception`

An exception representing a remote error returned from a goTenna device.

code = None
goTenna.constants.ErrorCodes member representing the error from the device

msg = None
String representing a human readable error message. May be empty.

exception `goTenna.constants.`**NotConnectedException**
Bases: `exceptions.Exception`

An exception raised when a method is called on a not-yet-connected device.

class `goTenna.constants.`**LAYER8_MESSAGE_TYPES**
Bases: `object`

Message types provided for convenience.

These are intended for use with the `goTenna.tlv.MessageTypeTLV` TLV for packing into message payloads. By default, the *TEXT_ONLY* type is used when a `goTenna.message.Payload` is built with `goTenna.message.Payload.text()`, and *SET_GROUP_GID* is used when a group invitation is created with `goTenna.message.Payload.group_invite()`.

These message types are used only at the application layer and are provided for convenience; they are not mandatory.

CUSTOM = '-1'

TEXT_ONLY = '0'

TEXT_AND_LOCATION = '1'

LOCATION_ONLY = '2'

LOCATION_REQUEST_ONLY = '3'

LOCATION_REQUEST_AND_TEXT = '4'

SET_GROUP_GID = '5'

LOG_ON_TEXT_ONLY = '6'

PING_TEXT_ONLY = '7'

FIRMWARE_PUBLIC_KEY_RESPONSE = '8'

USER_PUBLIC_KEY_RESPONSE = '9'

PUBLIC_KEY_REQUEST = '10'

FILE_TRANSFER = '11'

NET_RELAY_REQUEST = '12'

NET_RELAY_SUCCESS_RESPONSE = '13'

MESH_PUBLIC_KEY_REQUEST = '14'

MESH_PUBLIC_KEY_RESPONSE = '15'

FREQUENCY_SLOT_AND_TEXT = '16'

FREQUENCY_SLOT = '17'

TEXT_AND_MAP_PERIMETER = '18'

PERIMETER_ONLY = '19'

TEXT_AND_MAP_ROUTE = '20'

MAP_ROUTE_ONLY = '21'

TEXT_AND_CIRCLE = '22'

CIRCLE_ONLY = '23'

TEXT_AND_SQUARE = '24'

SQUARE_ONLY = '25'

GATEWAY_ADVERTISEMENT = '26'

BINARY_ONLY = '27'

classmethod valid(*to_check*)

goTenna.constants.**GID_MAX = 281474976710655**

The maximum possible value for a GID

goTenna.constants.**MAX_HOPS = 6**

The maximum number of times a message may hop on a goTenna network.

class goTenna.constants.**Bitrate**(*index*, *rate*, *timeouts*)
Bases: `object`

A bitrate with a rate in bits per second and index. This class should not be instantiated directly; it is present to define the allowed bitrates in *MASKS* and to configure `goTenna.settings.Bandwidth` objects.

index
The index of the bitrate.

timeouts
The list of configured timeouts as a list of `datetime.timedelta`

rate
The datarate as an int in bits/s.

class goTenna.constants.**Mask**(*index*, *width*, *allowed_bitrates*)
Bases: `object`

A channel mask with frequency in Hz, index, and allowed bitrates. This class should not be instantiated directly; it is present to define the allowed masks in *:py:attr:'MASKS* and to configure `goTenna.settings.Bandwidth` objects.

index
The index of the mask.

width
The width of the mask as an int in Hz.

allowed_bitrates
The allowed bitrates for this mask. A list of *goTenna.constants.Bitrate*.

bitrate_allowed(*bitrate*)
A query method that checks if a given bitrate is allowed at this mask.

Parameters bitrate (`goTenna.constants.Bitrate`) – The bitrate to check/

Returns Bool

class goTenna.constants.**Bandwidth**(*index*, *bandwidth*, *mask_idx*, *bitrate_idx*)
Bases: `object`

A bandwidth with frequency in kHz, index, and allowed masks. This class should not be instantiated directly; it is present to define the allowed bandwidth in *:py:attr:'BANDWIDTH_KHZ* and to configure *`goTenna.settings.RFSettings`* objects.

index
The index of the mask.

bandwidth
The bandwidth an float in kHz.

allowed_bandwidth
The allowed bandwidth. A list of *goTenna.constants.Bandwidth*.

goTenna.constants.**BANDWIDTH_KHZ = [<goTenna.constants.Bandwidth: 4.84 kHz>, <goTenna.const**
Valid channel masks and valid bitrates for those masks. Only values from the list are accepted when configuring a device.

goTenna.constants.**LOCATION_NAME_LENGTH_MAX = 32**
Maximum length for a location name to send in a `tlv.LocationNameTLV`

goTenna.constants.**LOCATION_TYPES = {'invalid': '0', 'location_periodic_sharing_auto_genera**
Types of locations we can encode. Best used with *`location_type()`* and *`location_type_name()`*.

`goTenna.constants.`**`location_type`**(*name*)

Look up a location type from its name.

Parameters **`name`** (`str`) – Key of *`LOCATION_TYPES`*

Raises **`KeyError`** – If `name` is not a key of *`LOCATION_TYPES`*.

`goTenna.constants.`**`location_type_name`**(*loc_type*)

Look up a location type name from the value.

Parameters **`loc_type`** – Value of *`LOCATION_TYPES`*

Raises **`KeyError`** – If `loc_type` is not in *`LOCATION_TYPES`*

`class` `goTenna.constants.`**`SDKLevels`**

`NORMAL = 0`

`SUPER = 1`

`class` `goTenna.constants.`**`OperationModes`**

`FIRST_VERSION_AVAILABLE = (0, 15, 17)`

`OFF = 0`

`NORMAL = 1`

`RELAY = 2`

`classmethod name`(*mode*)

The name of a member of this class

Parameters **`mode`** – Member of this class

Returns str The name of the mode

Raises **`KeyError`** – If the mode is not valid

`classmethod mode`(*name*)

The member of this class corresponding to the name.

Parameters **`name`** – The name of a member of this class (with any capitalization)

Returns int The member of the class

Raises **`KeyError`** – If the name is not valid

`class` `goTenna.constants.`**`GEO_REGION`**

Bases: `object`

A class defining acceptable values for geo region. Only the values defined in this class should be stored in GeoSettings.

`DICT = {1: 'REGION_NORTH_AMERICA', 2: 'REGION_EUROPE', 3: 'REGION_SOUTH_AFRICA', 4:`

`classmethod valid`(*value*)

A validator to ensure a given value is OK

`classmethod name`(*value*)

Get the name of a power level from its value

`classmethod value`(*name*)

Get the value of a powerlevel from a string name

4.6 goTenna.payload module

goTenna.payload: Classes defining message payloads

The classes defined here represent the payload part of messages. Some of the information contained in these payloads is required for routing, but the vast majority is user-specifiable and carries the content of the message.

The payload structure is made up of a base class, *Payload*, which defines all the metadata common to each payload type; and subclasses for each different kind of payload. Each payload type defines the message types (from *goTenna.constants.LAYER8_MESSAGE_TYPES*) that associate with it, and its own serialization and deserialization routines that handle the data specific to that payload.

A special subclass, *CustomPayload*, is provided to capture anything that cannot be parsed into one of the other subclasses of *Payload*. If parsing fails for any reason, the message is represented as containing a *CustomPayload* through which the raw contents of the payload, as byteslike, are accessible.

class goTenna.payload.**Payload**

Bases: `object`

A base class representing a message payload. Should not be instantiated directly; instead, should be built with one of the factory functions. Primarily exists to feed to the initializer or factory methods of `Message`.

The `Payload` type largely exists to be subclassed by specific kinds of payloads that know their message types and contents. To subclass `Payload`, a child must

- Override *serialize()* with a method that returns bytes representing the payload-specific parts of the payload: for instance, *TextPayload.serialize()* returns the serialized bytes of its *goTenna.tlv.payload_tlv.TextTLV*.
- Override *deserialize()* with a method that takes a list of TLVs, which will be the non-common parts of the payload received on the network.
- Have a property *payload_type* returning the appropriate member of `goTenna.constants.LAYER8_MESSAGE_TYPE` for the subclass. This is used when serializing messages to determine what message type should be encoded in the message.
- Have a class attribute `MESSAGE_TYPES` that is a tuple containing message types the class responds to. This should be a member of *goTenna.constants.LAYER8_MESSAGE_TYPES*. This is used when parsing received messages to select which subclass of *Payload* should be built.

Once these methods and attributes are defined, the class will work with *deserialize()* and *serialize()* and be automatically deserialized from incoming messages if the message's type matches one of the elements of `MESSAGE_TYPE`. If an exception is raised during deserialization, `Payload` will fall back to creating a *CustomPayload* that interprets the payload as bytes.

In general, it is not necessary to define custom payloads. The SDK comes with a number of `Payload` subclasses tailored to various different kinds of data, from locations to text to pings. The *CustomPayload* fallback can also be used as a container of bytes that are later parsed by the application.

The field containing the sender of the message is automatically updated by the driver when a payload is sent, unless it has previously been set via *Payload.set_sender()*. To explicitly set the sender of a payload to something other than the private GID configured in the driver, call *Payload.set_sender()* before sending the payload. If the sender is set when the payload is sent, and the driver is configured as a gateway, the sender is assumed to be external.

counter

The counter used to disambiguate identical messages

sender

Read-only property for accessing the message sender.

Returns goTenna.settings.GID The sender

time_sent
Return the time the payload was sent

Returns datetime.datetime The time. If the datetime is naive, it should be assumed to be in UTC.

valid
Property to check if all the common data is properly set and the payload is ready for transmission.

encrypted
Whether the payload is or should be encrypted

encryption_counter
The cryptographic encryption_counter used for the payload

__init__()
Build the payload.

This class should not be instantiated directly, and will never be seen when a message is received. Instead, one of the child classes of *Payload* should be used when sending, and will be received.

set_encryption(*should_encrypt*, *encryption_counter=0*, *encrypt_hook=None*)
Configure whether or not the palyoad should be encrypted

Parameters

- **should_encrypt** (*bool*) – Whether or not the payload should be encrypted. If True (or truthy), `encrypt_hook` should be specified.
- **encryption_counter** (*int*) – A encryption_counter to use for the encryption, if encryption is desired. This should fit in 16 bytes and should be different for every subsequent message between a specific pair of GIDs.
- **encrypt_hook** (*callable*) – A function to call to encrypt the contents of the payload. The hook is given the payload metadata. The form is .. function:: encrypt_hook(sender_gid, time_sent, encryption_counter, plaintext)

  param goTenna.settings.GID sender_gid The sender of the message

  param datetime.datetime time_sent The time the message was sent

  param int encryption_counter The payload encryption_counter, a 16-bit counter that changes with every meant-to-be-unique message sent from a specific sender to a specific destination

  param byteslike plaintext The message content to encrypt

set_sender(*sender*)
Set the sender of the payload.

Parameters sender (*None or goTenna.settings.GID*) – The sender. If `None`, reset and make the payload invalid.

Returns The object this method was called on, to allow chaining

The GID must be private. It may have a gateway set.

set_counter(*counter*)
Set the deduplication counter of the payload.

Parameters sender (*None or int*) – The counter. If `None`, reset and make the payload invalid. Should be in [0, 255].

Returns The object this method was called on, to allow chaining

set_sender_initials(*sender_initials*)
Set the sender initials of the payload.

Parameters **sender_initials** (None or `str`.) – The sender initials. If `None`, reset and make the payload invalid. Should be <= 4 bytes when encoded as UTF-8.

Returns The object this method was called on, to allow chaining

set_time_sent(*time_sent*)
Set the time the payload was sent.

Parameters **time_sent** (*None or datetime.datetime or int or time.struct_time*) – The time. If this is an aware `datetime.datetime`, it will be converted to UTC. If it is a naive `datetime.datetime`, an *int* representing a Posix timestamp, or a `time.struct_time` or similar tuple it is assumed to already be in UTC.

Returns The object this method was called on, to allow chaining

sender_initials
Accessor sender initials

Return str The initials

serialize()
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

to_tlv(*with_external_dest=None*)
Serialize the payload to a TLV-based bytestream.

hash_id
Return the hash used to associate this payload with its ACK.

This uses a special hash function duplicated in the firmware (which actually creates the ack message) and other SDKs.

classmethod from_tlv(*tlv*)
Parse the information present in a MessagePayloadTLV to a payload.

Parameters **goTenna.tlv.message_tlv.MessagePayloadTLV** – The TLV to build from.

payload_type
Get the payload type.

classmethod deserialize(*tlvs*)
Deserialize a payload from the payload-specific TLVs received on the network.

class goTenna.payload.**CustomPayload**(*contents*)
Bases: *goTenna.payload.Payload*

A payload that does not parse its input.

classmethod deserialize(*payload_bytes*)
Deserialize a CustomPayload from the payload-specific TLVs.

contents
The contents of the payload.

Returns bytes The contents, as bytes.

`__init__`(*contents*)
Build a CustomPayload.

Parameters **`content`** (`byteslike`) – The content of the payload, as a byteslike.

This class is intended for use if the types of message provided by this SDK are too restrictive. In that case, this payload allows the use of byteslike messages while still preserving required routing data.

`serialize`()
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`payload_type`
Get the payload type.

`class` `goTenna.payload.`**`TextPayload`**(*message*)
Bases: `goTenna.payload.Payload`

Payload containing only text.

`classmethod deserialize`(*tlvs*)
Deserialize a TextPayload from the payload-specific TLVs.

`__init__`(*message*)
Build a TextPayload.

This builds a TextPayload directly.

Parameters **`message`** (`str`) – The message to send.

`message`
The payload's message

`serialize`()
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`class` `goTenna.payload.`**`LocationPayload`**(*loc_id*, *loc_name*, *loc_latlong*, *loc_time*, *loc_type*, *accuracy*, *text*)
Bases: `goTenna.payload.Payload`

Payload containing location and perhaps also text.

`__init__`(*loc_id*, *loc_name*, *loc_latlong*, *loc_time*, *loc_type*, *accuracy*, *text*)
Build a LocationPayload.

Location payloads encapsulate - A numeric ID used for correlating updates to persistent positions - A human readable name - The location, as a (latitude, longitude) pair - The time the location was captured or is relevant for - The type of location - Optionally, the accuracy of the location - Optionally, a text component

Parameters

- **`loc_id`** (`int`) – The location ID.
- **`loc_name`** (`str`) – The human readable name. Should be <32 bytes encoded.
- **`float) loc_latlong`** (`tuple(float,)`) – The position of the location.
- **`loc_time`** (`datetime.datetime`) – A time that the position is relevant for.
- **`loc_type`** (`goTenna.constants.LOCATION_TYPES`) – The location type

- **accuracy** (*int or None*) – An integer number of meters between 0 and 65535 inclusive representing the accuracy of the location measurement
- **text** (*str or None*) – Text to send along with the message

map_id
The map ID associated with the location.

Returns int The ID

accuracy
The accuracy associated with the location, in meters

Returns int or None The accuracy, if available

name
The name associated with the location

Returns str The name

position
The actual position, as a (latitude, longitude) pair.

Returns tuple(float, float) The latitude and longitude

time
The time associated with the location.

Returns datetime.datetime The time, in UTC.

text
The text associated with the message, if there is any.

Returns str The text

loc_type
The type of location.

Returns str A value of *goTenna.constants.LOCATION_TYPES*

classmethod deserialize(*tlvs*)
Deserialize the Location payload from a list of TLVs.

:param list[goTenna.tlv.basic_tlv.TLV] A list of the TLVs in the message

Returns LocationPayload The initialized payload.

serialize()
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

class goTenna.payload.**LocationRequestPayload**(*request_gid*, *text*)
Bases: *goTenna.payload.Payload*

Payload containing a location request and perhaps also text.

classmethod deserialize(*tlvs*)
Deserialize a location request from a list of TLVs.

Parameters tlvs (*list[goTenna.tlv.basic_tlv.TLV]*) – The TLVs

Returns LocationRequestPayload The initialized payload

__init__(*request_gid*, *text*)
Build a LocationRequest.

Parameters

- **`request_gid`** (`goTenna.settings.GID`) – The request gid
- **`text`** (*`str`*) – An optional message to send along with the request. If no message is desired, set to `None`.

`serialize``()`
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`class` `goTenna.payload.`**`GroupGIDPayload`**`(`*group*`)`
Bases: *`goTenna.payload.Payload`*

Payload containing a group invitation with a group GID and member index.

`classmethod deserialize``(`*tlvs*`)`
Deserialize a GroupGIDPayload from a list of TLVs

Parameters **`tlvs`** (*`list[goTenna.tlv.basic_tlv.TLV]`*) – The TLVs.

Return GroupGIDPayload The initialized payload

`__init__``(`*group*`)`
Build a GroupGIDPayload.

The GroupGIDPayload is used as a group invitation, to specify the other members and the shared GID to a new participant. It contains the same information as found in a *`goTenna.settings.Group`* object and therefore requires one to initialize.

Parameters **`group`** (`goTenna.settings.Group`) – The group to pack into the payload.

Note: Because internal mechanisms require modification of the Group object, the argument is copied.

`set_sender``(`*sender*`)`
GroupGIDPayload override to ensure the sender is in the group.

Parameters **`sender`** (`goTenna.settings.GID`) – The sender of the message

`serialize``()`
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`group`
The group associated with the payload.

`class` `goTenna.payload.`**`PingPayload`**
Bases: *`goTenna.payload.Payload`*

Payload for a simple ping.

This payload does not carry any information beyond its existence; it is meant to serve as a lightweight way to see if a given destination is on the network. Though it can be used however the caller desires, the most frequent use is as a minimal payload for a private message, where the presence or absence of an acknowledgement in return indicates whether the recipient is on the network.

`classmethod deserialize``(`*tlvs*`)`
Deserialize a Ping from a list of TLVs

Parameters **tlvs** (*list[goTenna.tlv.basic_tlv.TLV]*) – The TLVs

Returns **goTenna.PingPayload** The initialized payload

__init__()

Build a ping payload.

serialize()

The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

class goTenna.payload.**PerimeterPayload**(*map_id*, *title*, *color*, *points*, *text*)

Bases: *goTenna.payload.Payload*

Payload containing perimeter and optionally text data

A perimeter encloses an area on a map. It contains a list of points (as (latitude, longitude) pairs), a title, a color, an ID, and optionally a message.

classmethod deserialize(*tlvs*)

Build a PerimeterPayload from a list of TLVs.

Parameters **tlvs** (*list[goTenna.tlv.basic_tlv.TLV]*) – The TLVs

Returns **PerimeterPayload** The initialized payload

__init__(*map_id*, *title*, *color*, *points*, *text*)

Build a PerimeterPayload

Parameters

- **map_id** (*int*) – A unique ID for the Perimeter object.
- **title** (*str*) – A short title to display for the Perimeter object.
- **color** – A color to specify for the Perimeter. The color should be a 4-tuple of (red, green, blue, alpha) channels that should either be `int` in [0, 255] or `float` in (0.0, 1.0) (where a higher value represents more intensity in the channel). For deserialization purposes, an int consisting of values backed as bytes in the order (alpha, red, green, blue) is also accepted.

Note: The color is transmitted on the network with 8 bits per pixel, even if it is specified as a float. Thus, the *PerimeterPayload.color* accessor always returns a tuple of ints, even if this object was created locally with floats.

Parameters

- **points** – The points making up the perimeter, as `list(tuple(float, float))` where the `tuple` elements are latitude and longitude, respectively. There may be up to 16 points in a perimeter.
- **text** – A message to send with the perimeter. This may be `None`, in which case no message will be sent. If not `None`, must be a stringlike that is less than 32 bytes long when encoded as UTF8.

serialize()

The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

map_id
The ID of the perimeter object.

color
The perimeter color.

This is a `tuple(int, int, int, int)`, where each element is in [0, 255] indicating the intensity of (in order) the red, green, blue, and alpha channels of the color. The color is presented as ints rather than floats to reflect the encoding; because this is how it is serialized, it cannot retain more than 8 bits per channel during transmission over the network, even if specified as floats.

title
The perimeter title. `str`.

points
The points making up the perimeter. `list(tuple(lat, long))`.

text
The message specified with the perimeter, if it exists. Otherwise `None`.

class goTenna.payload.**RoutePayload**(*map_id*, *title*, *color*, *points*, *text*)
Bases: *goTenna.payload.Payload*

Payload containing a route and optionally text data

Routes have very similar data to perimeters, but differ in their semantics. They represent a path on a map rather than the border of an area.

classmethod deserialize(*tlvs*)
Build a RoutePayload from a list of TLVs.

Parameters tlvs (*list[goTenna.tlv.basic_tlv.TLV]*) – The TLVs

Returns RoutePayload The initialized payload

__init__(*map_id*, *title*, *color*, *points*, *text*)
Build a RoutePayload

Parameters

- **map_id** (*int*) – A unique ID for the Route object.
- **title** (*str*) – A short title to display for the Route object.
- **color** – A color to specify for the Route. The color should be a 4-tuple of (red, green, blue, alpha) channels that should either be `int` in [0, 255] or `float` in (0.0, 1.0) (where a higher value represents more intensity in the channel). For deserialization purposes, an int consisting of values backed as bytes in the order (alpha, red, green, blue) is also accepted.

Note: The color is transmitted on the network with 8 bits per pixel, even if it is specified as a float. Thus, the *RoutePayload.color* accessor always returns a tuple of ints, even if this object was created locally with floats.

Parameters

- **points** – The points making up the route, as `list(tuple(float, float))` where the `tuple` elements are latitude and longitude, respectively. There may be up to 16 points in a route.

- **`text`** – A message to send with the route. This may be `None`, in which case no message will be sent. If not `None`, must be a stringlike that is less than 32 bytes long when encoded as UTF8.

`serialize``()`
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`map_id`
The ID of the route object.

`color`
The route color.

This is a `tuple(int, int, int, int)`, where each element is in [0, 255] indicating the intensity of (in order) the red, green, blue, and alpha channels of the color. The color is presented as ints rather than floats to reflect the encoding; because this is how it is serialized, it cannot retain more than 8 bits per channel during transmission over the network, even if specified as floats.

`title`
The route title. `str`.

`points`
The points making up the route. `list(tuple(lat, long))`.

`text`
The message specified with the route, if it exists. Otherwise `None`.

`class` `goTenna.payload.`**`CirclePayload`**`(`*map_id*, *title*, *center*, *radius*, *color*, *text*, **args*, ***kwargs*`)`
Bases: *`goTenna.payload.Payload`*

Payload containing a circle and optionally text data

`classmethod deserialize``(`*tlvs*`)`
Build a CirclePayload from a list of TLVs

Parameters **`tlvs`** (*`list[goTenna.tlv.basic_tlv.TLV]`*) – The list of TLVs

Return CirclePayload The initialized payload

`__init__``(`*map_id*, *title*, *center*, *radius*, *color*, *text*, **args*, ***kwargs*`)`
Build a CirclePayload.

Parameters

- **`map_id`** (*`int`*) – The ID for the circle.
- **`title`** (*`str`*) – The short title for the circle. This should be <32 bytes when encoded.
- **`float) center`** (*`tuple(float,`*) – The center of the circle, as a `tuple` of `(latitude, longitude)`.
- **`radius`** (*`float`*) – The radius of the circle.
- **`color`** – A color to specify for the circle. The color should be a 4-tuple of (red, green, blue, alpha) channels that should either be `int` in [0, 255] or `float` in (0.0, 1.0) (where a higher value represents more intensity in the channel). For deserialization purposes, an int consisting of values backed as bytes in the order (alpha, red, green, blue) is also accepted.

Note: The color is transmitted on the network with 8 bits per pixel, even if it is specified as a float. Thus, the `CirclePayload.color` accessor always returns a tuple of ints, even if this object was created locally with floats.

Parameters **`text`** (`str`) – A message that should go along with the circle. This is not mandatory. If `None`, no message will be sent.

`serialize``()`
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`map_id`
The map ID for the circle

`title`
The title of the circle

`center`
The center of the circle, as a `tuple(float, float)`

`radius`
The radius of the circle, in meters

`color`
The color of the circle, as a `tuple(int, int, int int)` describing the red, blue, green, and alpha channels. 0 is no intensity, 255 is most intense.

`text`
The text that came with the circle, or None.

`class` `goTenna.payload.`**`SquarePayload`**`(`*map_id*, *title*, *corners*, *color*, *text*`)`
Bases: `goTenna.payload.Payload`

Payload containing a square and optionally text data.

The Square is named as such for legacy reasons; it is actually a rectangle.

The Square is defined by three points, held internally as a list of (latitude, longitude) tuples.

The first two points define a line that defines one edge of the rectangle (the width). The distance between the third point and the width determines the length of the side of the rectangle normal to the width (the depth), and its position determines on which side of the width the rectangle extends.

`classmethod deserialize``(`*tlvs*`)`
Deserialize a SquarePayload from the payload-specific TLVs.

`__init__``(`*map_id*, *title*, *corners*, *color*, *text*`)`
Build the square.

Parameters

- **`map_id`** (`int`) – A unique ID for this shape.
- **`title`** (`str`) – A short title for the object.
- **`float)) corners`** (`list(tuple(float,`) – The three points that make up the square, as a length-3 iterable of (latitude, longitude) tuples.

- **`color`** – A color to specify for the square. The color should be a 4-tuple of (red, green, blue, alpha) channels that should either be `int` in [0, 255] or `float` in (0.0, 1.0) (where a higher value represents more intensity in the channel). For deserialization purposes, an int consisting of values backed as bytes in the order (alpha, red, green, blue) is also accepted.

Note: The color is transmitted on the network with 8 bits per pixel, even if it is specified as a float. Thus, the *`SquarePayload.color`* accessor always returns a tuple of ints, even if this object was created locally with floats.

Parameters **`text`** (*`str`*) – A message that should go along with the square. This is not mandatory. If `None`, no message will be sent.

`serialize``()`
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`title`
The title of the square

`map_id`
The ID of the square

`corners`
The three corners defining the shape

`color`
The color, as a tuple(float, float, float, float)

The tuple elements are the (r,g,b,a) channels in [0, 255].

`text`
The text associated with the square, if any

`class` `goTenna.payload.`**`FrequencySlotPayload`**`(`*rf_settings*, *name*, *callsign*, *slot_id*, *text*`)`
Bases: *`goTenna.payload.Payload`*

A payload communicating RF configuration information.

`rf_settings`
The *`goTenna.settings.RFSettings`* object associated with the payload.

Note: Because this payload is only capable of sending the channel mask, not the bitrate, the *`goTenna.settings.RFSettings`* object may not be identical to the one specified when the class was initialized. The payload defaults to selecting the maximum throughput (fastest) datarate for the specified mask.

`slot_id`
The slot id (a byteslike) associated with the payload.

`name`
The name (a `str`) associated with the payload.

`callsign`
The callsign (a `str`) associated with the payload.

`text`
The text (`str` or None) that might have been sent along with the payload.

classmethod deserialize(*tlvs*)
Build a FrequencySlotPayload from a list of TLVs

Parameters tlvs (*list[goTenna.tlv.basic_tlv.TLV]*) – The list of TLVs

Returns FrequencySlotPayload The initialized payload

__init__(*rf_settings*, *name*, *callsign*, *slot_id*, *text*)
Build a FrequencySlotPayload.

This payload is intended to share a complete frequency slot setup over the goTenna network.

Parameters rf_settings (goTenna.settings.RFSettings) – A valid *goTenna.settings.RFSettings* object to define the frequencies, transmit power, and channel mask the device should communicate on. This is the same object required to initialize a device.

Note: Only the channel mask is communicated over the network. The data rate is assumed to be the highest-throughput (fastest) data rate allowed by the mask. For this reason, the *goTenna.settings.RFSettings* object is copied and modified to have this datarate when it is passed in to this initializer.

Parameters

- **name** (*str*) – A human-readable name for the frequency slot. This should be less than 32 bytes when encoded as UTF-8.
- **callsign** (*str*) – A human-readable short callsign for the frequency slot. This should be less than 32 bytes when encoded as UTF-8.
- **slot_id** (*byteslike*) – An ID for the frequency slots. This should be a byteslike, 36 bytes or less long. This is intended to allow the use of UUIDs.
- **text** (*str*) – A message to send along with the frequency slots. This may be empty or None if there is no message to send.

serialize()
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

class goTenna.payload.**GatewayAdvertisementPayload**(*advertised_nodes*)
Bases: *goTenna.payload.Payload*

A payload advertising a gateway and what external nodes are available.

This payload can be used to advertise as a gateway with certain external GIDs available. Contacts specified in the payload should be sent as destinations with `via_gateway` set to the origin of this payload.

The list of nodes in this payload is not exhaustive; it is only a list of the numbers the gateway has chosen to advertise. Messages may arrive from the gateway with an origin that was not listed here.

The text sent along with each node should be human-readable and safe to display.

The advertised GIDs may have their `via_gateway` set, and may not; if they do not, it will be set as the sender (or sender's `via_gateway`, if present) set in the payload.

__init__(*advertised_nodes*)
Build the GatewayAdvertisementPayload.

Parameters **`advertised_nodes`** – A list of `tuple[int, str]` describing the node's ID and a short human readable description.

Raises

- **`TypeError`** – If `advertised_nodes` is not a list of `tuple[int,str]`
- **`ValueError`** – If the total length of the encoded payload is too long, or if a node ID is invalid (negative or cannot fit into 8 bytes)

`set_sender`(*sender*)
Override of set_sender for GatewayAdvertisementPayload that sets the gateway for its advertised nodes

`serialize`()
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`classmethod deserialize`(*tlvs*)
Deserialize a GatewayAdvertisementPayload from the payload-specific TLVs.

`class` `goTenna.payload.`**`BinaryPayload`**(*binary_data*)
Bases: *`goTenna.payload.Payload`*

A payload with binary data.

This payload can be used to send binary data.

`__init__`(*binary_data*)
Build the BinaryPayload.

Parameters **`binary_data`** – A binary data.

Raises **`ValueError`** – If the total length of the encoded payload is too long

`serialize`()
The base method for subclasses to serialize themselves

Payload subclasses should override this to return a bytestream containing their specific content. This method should not be called otherwise.

`classmethod deserialize`(*tlvs*)
Deserialize a BinaryPayload from the payload-specific TLVs.

4.7 goTenna.message module

Classes and functions for encoding, decoding, and manipulating high-level messages.

`class` `goTenna.message.`**`Message`**(*msgtype*, *hops*, *destination=None*, *membership=None*, *payload=None*, *encryptor=None*, *encryption_counter=0*, *is_repeated=False*)
Bases: `object`

A base class representing a message. Should not be instantiated directly; instead, should be built with one of the factory functions.

`__init__`(*msgtype*, *hops*, *destination=None*, *membership=None*, *payload=None*, *encryptor=None*, *encryption_counter=0*, *is_repeated=False*)
Build a message object. This should rarely be called in favor of the static factory methods.

Parameters

- **`msgtype`** – A key or value in *`goTenna.constants.MESSAGE_TYPES`*
- **`hops`** (*`int`*) – The number of times the message is allowed to mesh. Max of *`goTenna.constants.MAX_HOPS`*.
- **`destination`** (`goTenna.settings.GID` *or* *`None`*) – The destination for the message. Unnecessary (and ignored) if this message type does not require a specific destination.
- **`membership`** (*`int`*) – The membership index within the group of the sender, if this is a group message. Unnecessary (and ignored) if this is not a group message.
- **`payload`** (`goTenna.payload.Payload` *or* *`None`*) – The message payload. May be `None` or unspecified, or contain a payload that has not had its common data filled; however, until it is specified and the specified payload is fully valid (*`goTenna.payload.Payload.valid`* is `True`), the message will not be valid (*`Message.valid`* will be `False` and the message cannot be sent)
- **`encryptor`** – A function that will encrypt the payload of the message. The encryption function must be of the form: .. function:: encryptor(dest_gid, sender_gid, time_sent, encryption_counter, plaintext)

    param goTenna.settings.GID dest_gid The destination of the message

    param goTenna.settings.GID sender_gid The sender of the message (usually the local GID)

    param datetime.datetime time_sent The time the message is marked as sent

    param int encryption_counter A cryptographic encryption_counter

    param bytes plaintext The content to encrypt

    returns bytes The encrypted ciphertext

  The encryption function is given message metadata and content and should return the encrypted data.
- **`encryption_counter`** (*`int`*) – A 16-bit cryptographic encryption_counter that should change with every message sent between a given pair of GIDs.
- **`is_repeated`** (*`bool`*) – `True` if this message, or other messages like it, will be sent repeatedly - for instance, as automatically scheduled location updates. Specifying this value allows the goTenna network to better schedule and prioritize messages.

If the message has a destination other than the broadcast GID and `encryptor` is specified, the payload will be encrypted.

Raises

- **`ValueError`** – If one of the arguments has an invalid value - for instance, a negative hopcount or unknown message type.
- **`TypeError`** – If one of the arguments has an invalid type

`static broadcast`(*payload=None*, *hops=3*, *encryption_counter=0*, *is_repeated=False*)

Build a broadcast message containing a payload.

Parameters

- **`payload`** (*`None`* *or* `goTenna.payload.Payload`) – The contents of the message. If unspecified, the payload must be set later (e.g. by *`set_payload()`*).
- **`hops`** (*`int`*) – The maximum number of rebroadcasts on the network.

- **encryption_counter** (*int*) – The encryption counter. This should be different for every unique method.
- **is_repeated** (*bool*) – `True` if this message, or other messages like it, will be sent repeatedly - for instance, as automatically scheduled location updates. Specifying this value allows the goTenna network to better schedule and prioritize messages.

Returns Message The message, suitable for feeding into the SDK

counter
The disambiguating sequence number associated with the message.

This value depends on the payload associated with the message and will be None before the payload is set.

destination
The destination of the message.

static emergency (*payload=None*, *hops=3*, *encryption_counter=0*)
Build an emergency message containing a payload.

Parameters

- **payload** (*None or* `goTenna.payload.Payload`) – The contents of the message. If unspecified, the payload must be set later (e.g. by `set_payload()`).
- **encryption_counter** (*int*) – The encryption counter. This should be different for every unique method.

classmethod from_bytes (*bytestring*, *decrypt_hook=None*)
Parse a message from a bytestring as sent from the firmware via USB.

static group (*group_gid*, *index*, *payload=None*, *hops=3*, *encryptor=None*, *encryption_counter=0*, *is_repeated=False*)
Build a group message containing an arbitrary payload.

Parameters

- **group_gid** (`goTenna.settings.GID`) – The GID of the group
- **index** (*int*) – The index of the sender within the group
- **payload** (*None or* `goTenna.payload.Payload`) – The contents of the message. If unspecified, the payload must be set later (e.g. by `set_payload()`).
- **hops** (*int*) – The maximum number of rebroadcasts on the network.
- **encryptor** – A function that will encrypt the payload of the message. The encryption function must be of the form: .. function:: encryptor(dest_gid, sender_gid, time_sent, encryption_counter, plaintext)

  param goTenna.settings.GID dest_gid The destination of the message (the GID of the group)

  param goTenna.settings.GID sender_gid The sender of the message (usually the local GID)

  param datetime.datetime time_sent The time the message is marked as sent

  param int encryption_counter A cryptographic encryption_counter

  param bytes plaintext The content to encrypt

  returns bytes The encrypted ciphertext

The encryption function is given message metadata and content and should return the encrypted data.

- **encryption_counter** (*int*) – A 16-bit cryptographic encryption_counter that should change with every message sent between a given pair of GIDs.
- **is_repeated** (*bool*) – `True` if this message, or other messages like it, will be sent repeatedly - for instance, as automatically scheduled location updates. Specifying this value allows the goTenna network to better schedule and prioritize messages.

If `encryptor` is specified the payload will be encrypted before sending.

max_hops

The number of times the message may mesh.

Returns int The number of hops

membership

The membership index for the message.

message_type

The type of the message (group, broadcast, private).

Value of *goTenna.constants.MESSAGE_TYPES*

payload

The payload the message carries.

The payload will be a different subclass of *goTenna.payload.Payload* for different types of messages. If the payload has not yet been set, this method returns `None`.

There is a setter for this property, accepting an instance of *goTenna.payload.Payload*. The instance must be valid (*goTenna.payload.Payload.valid()* is `True`), and the setter only works if there is not already a valid payload stored in the message.

static private(*destination*, *payload=None*, *hops=3*, *encryptor=None*, *encryption_counter=0*, *is_repeated=False*)

Build a private message containing an arbitrary payload.

Parameters

- **destination** (`goTenna.settings.GID`) – The destination of the message
- **payload** (*None or* `goTenna.payload.Payload`) – The contents of the message. If unspecified, the payload must be set later (e.g. by *set_payload()*).
- **hops** (*int*) – The maximum number of rebroadcasts on the network.
- **encryptor** – A function that will encrypt the payload of the message. The encryption function must be of the form: .. function:: encryptor(dest_gid, sender_gid, time_sent, encryption_counter, plaintext)

  param goTenna.settings.GID dest_gid The destination of the message

  param goTenna.settings.GID sender_gid The sender of the message (usually the local GID)

  param datetime.datetime time_sent The time the message is marked as sent

  param int encryption_counter A cryptographic encryption_counter

  param bytes plaintext The content to encrypt

returns bytes The encrypted ciphertext

The encryption function is given message metadata and content and should return the encrypted data.

- **`encryption_counter`** (*`int`*) – A 16-bit cryptographic encryption_counter that should change with every message sent between a given pair of GIDs.
- **`is_repeated`** (*`bool`*) – `True` if this message, or other messages like it, will be sent repeatedly - for instance, as automatically scheduled location updates. Specifying this value allows the goTenna network to better schedule and prioritize messages.

If `encryptor` is specified the payload will be encrypted before sending.

`sender`
The sender of the message.

This value depends on the payload and will be `None` before the payload is set.

`set_max_hops`(*hops*)
Set the number of times the message may mesh.

Parameters **`hops`** (*`int`*) – The number of hops

`set_payload`(*payload*)
Set the payload associated with the message.

Parameters **`payload`** (`goTenna.payload.Payload` *or* *`None`*) – The Payload to set, if any. Should be a subclass of *`goTenna.payload.Payload`*.

`time_sent`
The time the message was sent.

This value depends on the payload associated with the message and will be `None` before the payload is set.

Returns datetime.datetime The time

`to_bytes`(*sdk_token*)
Write the message to a bytestring suitable for sending via USB to a connected device.

Parameters **`sdk_token`** (*`bytes`*) – The SDK token of this driver instance

`to_tlvs`(*sdk_token*)
Serialize a message into a list of `goTenna.basic_tlv.TLV` which can then by serialized to bytes.

Parameters **`sdk_token`** (*`bytes`*) – The SDK token of this driver instance

Raises **`ValueError`** – If the message is not fully initialized, i.e. if the payload has not been set. This can be checked with *`Message.valid`* - if that is `True`, the message is ready to go.

`update_encryption_counter`(*new_counter_val*)
Update the encryption counter for the message.

If a message is resent on the network because an acknowledgement for it never arrived, when it is resent it should have its encryption counter changed. This prevents the network from deduplicating the message.

If this message is not encrypted, changing the counter still prevents deduplication.

Parameters **`new_counter_val`** (*`int`*) – The new value to use. Should be between 0 and 65535 and different from the currently-stored counter value.

`valid`
Return whether the message is valid and ready to send.

4.8 goTenna.tlv.payload_tlv module

TLVs intended to be placed in message payloads

The TLVs contained here are for encoding data to send via the goTenna network. In general, they should not be used directly; rather, message payloads should be created with the high level factory classmethods on `goTenna.message.Payload`, such as `goTenna.message.Payload.text()`.

class `goTenna.tlv.payload_tlv.`**`LatLongBase`**(*latitude*, *longitude*)

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV base class that just holds a latitude/longitude pair

`__init__`(*latitude*, *longitude*)

Build the object.

Parameters

- **`latitude`** (`float`) – The latitude to store
- **`longitude`** (`float`) – The longitude to store

Raises **`TypeError`** – If one of latitude or longitude is not convertible to float.

`latitude`

The latitude, as a float

`longitude`

The longitude, as a float

`position`

The position, as a (latitude, longitude) pair

`serialize`()

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

classmethod **`deserialize`**(*value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

class `goTenna.tlv.payload_tlv.`**`RequestGIDTLV`**(*gid*)

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding a request for a GID

`gid`

The GID associated with the TLV

`__init__`(*gid*)

Build a RequestGIDTLV.

This is usually used as the source value for some sort of request message. As such, it should be a private GID.

Parameters **`gid`** (`goTenna.settings.GID`) – The GID to pack.

Raises

- **TypeError** – If `gid` is not a *goTenna.settings.GID*.
- **ValueError** – If `gid` is not a private GID.

classmethod deserialize(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class goTenna.tlv.payload_tlv.**MapIDTLV**(*map_id*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV holding the ID of the map a message contains

map_id
The map id contained by the TLV

__init__(*map_id*)
Build a MapIDTLV.

Map IDs are 64-bit integers intended to uniquely identify generic map objects such as routes or circles across multiple communicating users. They should be generated with `uuid.uuid4()` or `os.urandom()` to ensure global uniqueness. However, by the time they are given to this TLV, they should be coalesced to integers.

Parameters **map_id** (*int*) – The ID.

Raises **TypeError** – IF `map_id` is not an int.

classmethod deserialize(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class goTenna.tlv.payload_tlv.**LocationNameTLV**(*text*)
Bases: `goTenna.tlv.basic_tlv.BasicTextTLV`

A TLV for holding a location name

name
The location name associated with the TLV

max_length
The maximum length of the TLV

class goTenna.tlv.payload_tlv.**LocationLatitudeTLV**(*latitude*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding a latitude

latitude
The latitude associated with the TLV

__init__(*latitude*)
Build a latitude TLV.

Parameters **latitude** (*float*) – The latitude to encode.

Raises **TypeError** – If `latitude` is not a float.

classmethod deserialize(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class goTenna.tlv.payload_tlv.**LocationLongitudeTLV**(*longitude*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding longitude

longitude
The longitude associated with the TLV

__init__(*longitude*)
Build a longitude TLV.

Parameters **float** – longitude: The longitude to encode.

Raises **TypeError** – If `longitude` is not a float.

classmethod deserialize(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class goTenna.tlv.payload_tlv.**LocationGPSTimestampTLV**(*timestamp*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding a GPS timestamp

timestamp
The naive UTC timestamp associated with the TLV

__init__(*timestamp*)
Build a GPS timestamp.

This object holds the time internally as a `datetime.datetime` object.

Parameters **`timestamp`** (*`datetime.datetime`*) – The timestamp to hold. If it is naive, it will be stored as is. If it is aware, it will be converted to UTC in preparation for serialization. All deserialized timestamps are assumed to be UTC.

`classmethod deserialize`(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

`serialize`()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

`class` `goTenna.tlv.payload_tlv.`**`LocationTypeTLV`**(*location_type*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding a location type

`location_type`
The location type associated with the TLV

`__init__`(*location_type*)
Build a location type.

Parameters **`location_type`** (*`str`*) – A key of *`goTenna.constants.LOCATION_TYPES`* to send.

Raises **`KeyError`** – if `location_type` is not a valid location type.

`classmethod deserialize`(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

`serialize`()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

`class` `goTenna.tlv.payload_tlv.`**`LocationSharingFrequencyTLV`**(*frequency*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV to specify how frequently a location is shared

`frequency`
The sharing frequency associated with the TLV

`__init__`(*frequency*)
Build a location sharing frequency.

Parameters **`frequency`** (*`int`*) – The sharing a frequency, a key of `goTenna.constants.SHARING_FREQUENCIES`

Raises `KeyError` – If `frequency` is not in `goTenna.constants.SHARING_FREQUENCIES`

`serialize`()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

`classmethod deserialize`(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

`class` `goTenna.tlv.payload_tlv.`**`LocationAccuracyTLV`**(*accuracy*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding the accuracy of the location message

`accuracy`
The accuracy value held by the TLV, in meters

`__init__`(*accuracy*)
Build a location sharing accuracy TLV

Parameters **`accuracy`** (*`int`*) – The accuracy of the measurement, in meters.

Raises

- `TypeError` – If `accuracy` cannot be converted to `int`
- `ValueError` – If `accuracy` is less than 0 or larger than 65535

`classmethod deserialize`(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

`serialize`()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

`class` `goTenna.tlv.payload_tlv.`**`LocationDataTLV`**(*tlvs*)
Bases: `goTenna.tlv.basic_tlv.BasicContainerTLV`

A TLV for holding location data

`class` `goTenna.tlv.payload_tlv.`**`GroupGIDTLV`**(*gid*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding a group's GID

`gid`
Accessor for parsed/specified GID

`__init__`(*gid*)
Build the TLV

Parameters **gid** (*long or goTenna.settings.GID*) – The GID of the group

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

classmethod deserialize(*tlv_value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

class goTenna.tlv.payload_tlv.**GroupMemberListTLV**(*members*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding a group member list

members
The members of the group

__init__(*members*)
Build the TLV

Parameters **members** (*list[goTenna.goTenna.settings.GID]*) – The members of the group. Ordering should be preserved across instances of the group. This list is assumed to start at index 1, and should not include the creator of the group.

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

classmethod deserialize(*tlv_value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

class goTenna.tlv.payload_tlv.**GroupSharedSecretTLV**(*secret*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding a group shared secret

secret
The secret contained in the TLV.

__init__(*secret*)
Build the TLV.

Parameters **secret** (*byteslike*) – The shared secret.

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

classmethod deserialize(*tlv_value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

class goTenna.tlv.payload_tlv.**EncryptionInfoTLV**(*sender*, *timestamp*, *encrypted*, *counter*, *message_id*)

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for a message encryption header

__init__(*sender*, *timestamp*, *encrypted*, *counter*, *message_id*)

Create an EncryptionInfoTLV.

Parameters

- **sender** (`goTenna.settings.GID`) – The sender of the message.
- **timestamp** (*int or datetime.datetime or tuple*) – The time to associate with the message. Used for ordering.
- **encrypted** (*bool*) – Whether the message is encrypted. Note: This value is currently ignored when packing a TLV, and is present in this constructor for representing the value read from a device.
- **counter** (*int*) – The index into the shared secret that came along with this message. Note: This value is currently ignored when packing a TLV, and is present in this constructor for representing the value read from a device.
- **message_id** (*int*) – The message ID, for supporting message resend deduplication. Unused here.

time

A property returning the timestamp as a proper datetime.datetime.

timestamp

A property returning the timestamp as a Posix epoch timecode.

classmethod deserialize(*tlv_value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize()

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class goTenna.tlv.payload_tlv.**FrequencySlotDataBandwidthTLV**(*bw*)

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for sending frequency slots.

Currently this only holds the channel mask width, and does not specify a bitrate. When the SDK parses one of these TLVs from the message, it will default to using the higher bitrate associated with the mask.

serialize()

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

classmethod deserialize(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

class goTenna.tlv.payload_tlv.**FrequencySlotMaxPowerTLV**(*power*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for frequency slot power

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

classmethod deserialize(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

class goTenna.tlv.payload_tlv.**FrequencySlotFrequencyListTLV**(*control_freqs*, *data_freqs*)
Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV holding channel frequencies

__init__(*control_freqs*, *data_freqs*)
Build the frequency list.

Parameters

- **control_freqs** (*list[int]*) – A list of control frequencies. Each must pass *goTenna.settings.RFSettings.validate_freq()*. There should be at least one and fewer than three.
- **data_freqs** (*list[int]*) – A list of data frequencies. Each must pass *goTenna.settings.RFSettings.validate_freq()*. There should be at least one and less than 15.

Raises

- **TypeError** – If any frequency or list of frequencies is the wrong type.
- **ValueError** – If any frequency is not valid.

data_freqs
The data frequencies. `list[int]`.

control_freqs
The control frequencies. `list[int]`.

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

classmethod deserialize(*value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

class `goTenna.tlv.payload_tlv.`**FrequencySlotNameTLV**(*text*)

Bases: `goTenna.tlv.basic_tlv.BasicTextTLV`

A TLV holding the name of a slot set.

name

The title associated with the TLV

class `goTenna.tlv.payload_tlv.`**FrequencySlotIDTLV**(*id_bytes*)

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV holding an ID for a slot set.

The ID is intended to be a UUID and thus should be byteslike.

__init__(*id_bytes*)

Build the frequency slot ID.

Parameters **id_bytes** (*byteslike*) – The byteslike ID (e.g. a UUID). 36 bytes max.

Raises

- **TypeError** – If `id_bytes` is not byteslike.
- **ValueError** – If `id_bytes` is more than 36 bytes long, or empty

serialize()

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

classmethod deserialize(*value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

class `goTenna.tlv.payload_tlv.`**FrequencySlotCallSignTLV**(*text*)

Bases: `goTenna.tlv.basic_tlv.BasicTextTLV`

A TLV for a frequency slots short call sign.

callsign

The description for the frequency slots.

class `goTenna.tlv.payload_tlv.`**FrequencySlotDataTLV**(*tlvs*)

Bases: `goTenna.tlv.basic_tlv.BasicContainerTLV`

The container TLV for frequency slots.

class `goTenna.tlv.payload_tlv.`**PerimeterTitleTLV**(*text*)

Bases: `goTenna.tlv.basic_tlv.BasicTextTLV`

A TLV for human readable perimeter titles

title
The title associated with the TLV

class goTenna.tlv.payload_tlv.**PerimeterPointsTLV**(*points*)
Bases: goTenna.tlv.basic_tlv.TLV

A TLV for the points making up a perimeter

points
The sequence of map points held by the tlv

__init__(*points*)
Build the PerimeterPointsTLV.

This TLV encompasses a list of points as (latitude, longitude) points It can hold up to 8 such points.

Parameters **points** (*list[tuple[float, float]]*) – The points as a list of (lat, long) pairs.

Raises **ValueError** – If there are too many points or if the points are malformed.

classmethod deserialize(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class goTenna.tlv.payload_tlv.**TextTLV**(*text*)
Bases: goTenna.tlv.basic_tlv.BasicTextTLV

A TLV for holding message text

class goTenna.tlv.payload_tlv.**PerimeterColorTLV**(*red=None, green=None, blue=None, alpha=None, color_int=None, color_tuple=None*)
Bases: goTenna.tlv.basic_tlv.BasicColorTLV

class goTenna.tlv.payload_tlv.**PerimeterDataTLV**(*tlvs*)
Bases: goTenna.tlv.basic_tlv.BasicContainerTLV

A TLV encapsulating perimeter data.

This is used as a container for other perimeter-related TLVs. Only certain TLVs are allowed to be present in PerimeterData; they are the classes listed in `PerimeterDataTLV.ACCEPTABLE_CONTENTS`.

class goTenna.tlv.payload_tlv.**RouteTitleTLV**(*text*)
Bases: goTenna.tlv.basic_tlv.BasicTextTLV

A TLV for human readable route titles

title
The title of the route

class goTenna.tlv.payload_tlv.**RoutePointsTLV**(*points*)
Bases: goTenna.tlv.basic_tlv.TLV

A TLV for the points making up a route

points
The points contained in the TLV

`__init__`(*points*)
Build a RoutePoints TLV.

Parameters **float]] points** (*list[tuple[float,*) – The points, as (latitude, longitude) pairs. Maximum 8.

Raises

- **ValueError** – If there are too many points or if any points are malformed.
- **TypeError** – If anything is the wrong type.

classmethod deserialize(*value*)
A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize()
A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class goTenna.tlv.payload_tlv.**RouteColorTLV**(*red=None*, *green=None*, *blue=None*, *alpha=None*, *color_int=None*, *color_tuple=None*)
Bases: `goTenna.tlv.basic_tlv.BasicColorTLV`

A TLV for the color the route should be displayed with

class goTenna.tlv.payload_tlv.**RouteDataTLV**(*tlvs*)
Bases: `goTenna.tlv.basic_tlv.BasicContainerTLV`

A TLV encapsulating route data

acceptable_contents
The contents of the TLV.

class goTenna.tlv.payload_tlv.**CircleTitleTLV**(*text*)
Bases: `goTenna.tlv.basic_tlv.BasicTextTLV`

A TLV containing the name of a circle.

title
The title of the route

class goTenna.tlv.payload_tlv.**CircleCenterTLV**(*latitude*, *longitude*)
Bases: *goTenna.tlv.payload_tlv.LatLongBase*

A GPS point marking the center of a circle.

`__init__`(*latitude*, *longitude*)
Build the CircleCenterTLV.

Parameters

- **latitude** (*float*) – The latitude of the center.
- **longitude** (*float*) – The longitude of the center.

Raises **TypeError** – If one of the params is not convertible to float.

class goTenna.tlv.payload_tlv.**CircleRadiusTLV**(*radius*)

Bases: goTenna.tlv.basic_tlv.TLV

The radius of a circle, in meters

__init__(*radius*)

Build the radius TLV.

Parameters **radius** (*float*) – The radius, in meters.

Raises **TypeError** – If the radius is not convertible to float.

radius

The radius of the circle, in meters.

serialize()

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use TLV.to_bytes().

classmethod deserialize(*value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use from_bytes() or multiple_from_bytes()

class goTenna.tlv.payload_tlv.**CircleColorTLV**(*red=None*, *green=None*, *blue=None*, *alpha=None*, *color_int=None*, *color_tuple=None*)

Bases: goTenna.tlv.basic_tlv.BasicColorTLV

A TLV for the color a circle should be displayed as

class goTenna.tlv.payload_tlv.**CircleDataTLV**(*tlvs*)

Bases: goTenna.tlv.basic_tlv.BasicContainerTLV

A TLV encapsulating circle data

class goTenna.tlv.payload_tlv.**SquareTitleTLV**(*text*)

Bases: goTenna.tlv.basic_tlv.BasicTextTLV

A TLV containing the name of a circle.

title

The title of the route

class goTenna.tlv.payload_tlv.**SquareCornerOneTLV**(*latitude*, *longitude*)

Bases: *goTenna.tlv.payload_tlv.LatLongBase*

A GPS point marking the first corner of a square.

__init__(*latitude*, *longitude*)

Build the SquareCornerOneTLV.

Parameters

- **latitude** (*float*) – The latitude of the corner
- **longitude** (*float*) – The longitude of the corner

Raises **TypeError** – If one of the params is not convertible to float.

class `goTenna.tlv.payload_tlv.`**`SquareCornerTwoTLV`**(*latitude*, *longitude*)

Bases: *`goTenna.tlv.payload_tlv.LatLongBase`*

A GPS point marking the second corner of a square.

`__init__`(*latitude*, *longitude*)

Build the SquareCornerTwoTLV.

Parameters

- **`latitude`** (*`float`*) – The latitude of the corner
- **`longitude`** (*`float`*) – The longitude of the corner

Raises **`TypeError`** – If one of the params is not convertible to float.

class `goTenna.tlv.payload_tlv.`**`SquareDepthTLV`**(*latitude*, *longitude*)

Bases: *`goTenna.tlv.payload_tlv.LatLongBase`*

A GPS point marking a point along the opposite side of the square as corners 1 and 2

`__init__`(*latitude*, *longitude*)

Build the SquareDepth.

Parameters

- **`latitude`** (*`float`*) – The latitude of the depth point
- **`longitude`** (*`float`*) – The longitude of the depth point

Raises **`TypeError`** – If one of the params is not convertible to float.

class `goTenna.tlv.payload_tlv.`**`SquareColorTLV`**(*red=None*, *green=None*, *blue=None*, *alpha=None*, *color_int=None*, *color_tuple=None*)

Bases: `goTenna.tlv.basic_tlv.BasicColorTLV`

A TLV for the color a square should be displayed as

class `goTenna.tlv.payload_tlv.`**`SquareDataTLV`**(*tlvs*)

Bases: `goTenna.tlv.basic_tlv.BasicContainerTLV`

A TLV encapsulating square data

class `goTenna.tlv.payload_tlv.`**`PublicKeyDataTLV`**(*key_bytes*)

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for public key data

`__init__`(*key_bytes*)

Build a PublicKeyDataTLV.

Parameters **`key_bytes`** (*`bytearray or bytes depending on Python version.`*) – The content of the key. Should be less than goTenna.constants.MAXLENGTH long. This will be serialized directly.

classmethod **`deserialize`**(*value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

`serialize`()

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class `goTenna.tlv.payload_tlv.`**`SenderInitialsTLV`**`(`*text*`)`

Bases: `goTenna.tlv.basic_tlv.BasicTextTLV`

A TLV for holding sender initials

class `goTenna.tlv.payload_tlv.`**`MessageTypeTLV`**`(`*msgtype*`)`

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV for holding a message type string

classmethod deserialize`(`*tlv_value*`)`

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize`()`

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class `goTenna.tlv.payload_tlv.`**`ExternalDestinationTLV`**`(`*destination*`)`

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV specifying a non-goTenna destination for a message (e.g. through a gateway)

classmethod deserialize`(`*tlv_value*`)`

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize`()`

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class `goTenna.tlv.payload_tlv.`**`ExternalOriginTLV`**`(`*origin*`)`

Bases: `goTenna.tlv.basic_tlv.TLV`

A TLV specifying a non-goTenna origin for a message (e.g. from a gateway)

classmethod deserialize`(`*tlv_value*`)`

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use `from_bytes()` or `multiple_from_bytes()`

serialize`()`

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use `TLV.to_bytes()`.

class goTenna.tlv.payload_tlv.**AdvertisedExternalAddressTLV**(*gid*, *description*)

Bases: goTenna.tlv.basic_tlv.TLV

A TLV specifying the ID and human readable description for a node external to the goTenna network.

The description must be less than 218 bytes encoded, though if more than one TLV is to fit in a payload it should probably be shorter than that.

__init__(*gid*, *description*)

Build the TLV.

Parameters

- **address** (*gid*) – The external GID. Assumed to be external, so it does not need to have its via_gateway explicitly set.
- **description** (*str*) – The human-readable description of the address. Must be less than 218 bytes encoded, though if more than this TLV is to go in a message it should probably be less.

Raises **ValueError** – If the address is invalid, or the description is invalid.

serialize()

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use TLV.to_bytes().

classmethod deserialize(*tlv_value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use from_bytes() or multiple_from_bytes()

class goTenna.tlv.payload_tlv.**BinaryTLV**(*binary*)

Bases: goTenna.tlv.basic_tlv.TLV

A TLV specifying the ID and human readable description for a node external to the goTenna network.

The description must be less than 218 bytes encoded, though if more than one TLV is to fit in a payload it should probably be shorter than that.

__init__(*binary*)

Build the TLV.

Raises **ValueError** – If the length of the binary is too long.

serialize()

A do-nothing base method for building TLV data from a TLV class

Should be implemented by subclasses to build the payload only. This method should not be called directly; instead, use TLV.to_bytes().

classmethod deserialize(*tlv_value*)

A do-nothing base classmethod for parsing a TLV from serialized data.

Should be implemented by subclasses to parse data from the value. Subclasses may assume that this method will only be called if it is in fact data for that subclass.

Should not be called directly; instead, use from_bytes() or multiple_from_bytes()

goTenna.tlv.payload_tlv.**tlvclass**

alias of *goTenna.tlv.payload_tlv.MapIDTLV*

4.9 goTenna.storage module

Classes and functions for the goTenna storage layer.

This module defines both the high level storage interface to which storage objects given to `goTenna.driver.Driver` must adhere and a concrete implementation that stores data in a local file encrypted with the SDK token.

class goTenna.storage.**EncryptedFileStorage**(*sdk_key*, *filename='.goTenna'*, *work_factor=100000*)

Bases: `goTenna.storage.StorageInterface`

A storage implementation that stores data in a file, encrypted with the current SDK key.

__init__(*sdk_key*, *filename='.goTenna'*, *work_factor=100000*)

Construct a `EncryptedFileStorage`.

The file storage is encrypted, since it stores key material for any private GID used with the SDK.

Parameters

- **sdk_key** (`str`) – The SDK key being used with this SDK instance. The key is used as an input to the encryption used in the file storage, ensuring that other users of the goTenna SDK cannot read data encrypted with a different key.
- **filename** (`str`) – The file to use for storage. By default this is a dotfile in the directory in which Python is invoked and loads the SDK.
- **work_factor** (`int`) – The work factor to use for the key derivation function. Higher values make the encryption of the config more resistant to bruteforcing, but may take a long time on non-powerful systems.

Raises

- **ValueError** – If the specified file exists (or can be created) but cannot be decrypted, either because it does not contain encrypted data or because it was encrypted with a different SDK key.
- **OSError** – If the specified file does not exist and cannot be created (for instance because the specified directory does not exist)

load(*gid*)

Load the data stored for the given private GID.

When this function is called, it should always read from the nonvolatile storage; however, it does not need to be the only place the implementation reads from the nonvolatile storage.

Parameters **gid** (`goTenna.settings.GID`) – The private GID to load the configuration for.

Returns bytes The encrypted data

remove_records(*gid*)

Remove the records for the given private GID.

This method can be used to delete the stored private key for the GID and any public keys for links to other private GIDs and groups that the private key is in.

This method should not be called on a GID that is currently active.

store()

Update the stored data for the GID specified in `load()`

By the time this function returns, the settings should be stored persistently in case the application terminates. It does not need to be the only place where the backend writes to nonvolatile storage.

class `goTenna.storage.`**`StorageInterface`**

Bases: `object`

A class defining how a goTenna SDK compliant storage class should work.

The primary job of the storage interface is to store the known public keys for other private GIDs, and the members and shared secrets for known groups. Each of these are only valid for a specific private GID with which the SDK is configured.

Implementers of this interface must provide

- the backend methods *`load()`*, *`store()`*, and *`remove_records()`* which provide the storage backend. *`load()`* and *`store()`* load and store data in nonvolatile storage respectively. When these methods are called, they must read from the nonvolatile storage and write to it, but the implementation can read and write to the backend at every times if desired. *`remove_records()`* deletes all records for the GID and can be used to erase key material for a GID that is no longer used.
- the frontend properties *`groups`*, *`link_pubkeys`*, and *`local_key`* with setter, which provide the interface for the rest of the SDK to access data in the same structured way no matter the backend data structures.

`encryption_counters`

A property for the stored per-destination cryptographic encryption_counters.

The *`StorageInterface`* stores a history of the encryption_counters used for each destination GID, whether group or private. This is used to reject duplicate messages and ensure that sent messages are not rejected as duplicates. The storage of these encryption_counters should preserve the order in which they were added.

Return dict[gid]->list[int] A mapping between a destination GID and the encryption_counter history for it.

`groups`

A property for the stored groups.

The *`StorageInterface`* stores data representing groups that the local GID is part of.

Returns dict A dict mapping group GIDs (as *`goTenna.settings.GID`*) to the *`goTenna.settings.Group`* object for that group.

`link_pubkeys`

A property for the stored public keys for other devices.

When encryption is enabled, the SDK automatically exchanges public keys with any device that is sent a private message and stores them in this object.

Returns dict A dict mapping other private GIDs (as *`goTenna.settings.GID`*) to the public key for that GID (as `byteslike`).

`load`(*gid*)

Load the data stored for the given private GID.

When this function is called, it should always read from the nonvolatile storage; however, it does not need to be the only place the implementation reads from the nonvolatile storage.

Parameters **`gid`** (`goTenna.settings.GID`) – The private GID to load the configuration for.

Returns bytes The encrypted data

`local_key`

Returns byteslike The encoded private key for the GID this object was initialized with.

`remove_records`(*gid*)

Remove the records for the given private GID.

This method can be used to delete the stored private key for the GID and any public keys for links to other private GIDs and groups that the private key is in.

This method should not be called on a GID that is currently active.

`set_local_key`

Returns byteslike The encoded private key for the GID this object was initialized with.

`store`()

Update the stored data for the GID specified in `load()`

By the time this function returns, the settings should be stored persistently in case the application terminates. It does not need to be the only place where the backend writes to nonvolatile storage.

4.10 Logging

SDK level logging is provided through the `logging` standard Python module. The SDK modules and classes define loggers with the standard dot-separated namespacing hierarchy, and do not instantiate any default log handlers. SDK logging may be controlled by setting properties of the *goTenna* log hierarchy.

PYTHON MODULE INDEX